Вторников А. А.

СТЕК, или ПУТЕШЕСТВИЕ ТУДА И ОБРАТНО



Москва, 2017

УДК 004.6:004.42
ББК 32.972
В87

Вторников А. А.
В87 Стек, или Путешествие туда и обратно. – М.: ДМК Пресс, 2017. – 140 с.: ил.

ISBN 978-5-97060-517-2

Книга посвящена простой и удивительно элегантной структуре данных – стеку. Описаны скобочные структуры, подпрограммы (в том числе рекурсивные), передача параметров, разбор и вычисление выражений, распознавание последовательностей символов. Также рассмотрено описание устройства и реализация простой, но достаточно мощной стековой машины; приведены многочисленные примеры программ, а также список задач, в том числе нетривиальных. На сайте издательства dmkpress.com содержатся дополнительные материалы, среди которых исходные коды простого транслятора стековой машины (на языке Java).

Издание предназначено прежде всего пытливым старшеклассникам, студентам вузов, а также тем, для кого программирование – хобби.

УДК 004.6:004.42
ББК 32.972

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-97060-517-2

© Вторников А. А., 2017
© Оформление, издание, ДМК Пресс, 2017

Содержание

Вместо предисловия 5

Введение 6

Часть I. Задачи, приводящие к стеку 10

Скобочные структуры: элементарный случай 10
Стек: знакомство 18
Скобочные структуры: общий случай 24
Подпрограммы: постановка задачи 28
Подпрограммы: появляется стек 41
Подпрограммы: рекурсия 46
Знакомая незнакомка: арифметика 55
  Алгоритм трансляции выражений 61
Стек как вычислительное устройство 67
Стековая машина 70
Подпрограммы: передача параметров 75
Стек и грамматики 89
Заключение 98

Часть II. От слов – к делу 99

Расширенная стековая машина 99
Первая программа 105
Как работает транслятор 108
  Класс ToyStackMachine.java 109
  Класс OpcodeTable.java 109
  Класс SymbolTable.java 111
  Класс Assembler.java 111
  Класс StackMachine.java 113
  Расширение системы команд 114
Примеры программ 115
  Суммирование последовательности чисел 115
  Сложение двух чисел (вариант 1) 116
  Сложение двух чисел (вариант 2) 117

Сложение последовательности чисел в памяти 118
Факториал 119
Вывод текста 120
Программа-шутка 121
Указания по программированию 122
К вершинам мастерства 122
Заключение 123

Библиография 124

Приложение А. Способы реализации стеков 126
Реализация стека на основе массива 126
Реализация стека на основе связанного списка 128

Приложение В. Язык Forth 131

Приложение С. Стек и современные языки программирования 136

Приложение D. Исходные коды транслятора и интерпретатора 138

Вместо предисловия

Несмотря на то что почти всякая книга по программированию начинается с предисловия, оно почти всегда пишется последним. Для автора книги это последний шанс оправдать (хотя бы для самого себя) и мелкость темы, и убогость стиля, и отсутствие литературного таланта.

Будем честными: предисловия пишут не для того, чтобы их читали. Это единственное место, где можно попытаться убедить потенциального читателя потратить свое время и свои деньги на книгу до того, как он поймет, что ошибся. Косвенно переложив тем самым ответственность за выбор на него.

Автор убежден, что это не имеет ничего общего с моралью, и поэтому отказывается от ложных оправданий и пустых обещаний. Да, он писал книгу не без удовольствия, но вряд ли это может служить весомой рекомендацией. Единственный совет, который он может предложить, – обратиться к оглавлению, выбрать любой раздел и попробовать почитать. Первое впечатление обычно и самое верное.

Впрочем, одно достоинство у предисловия все-таки есть. В нем автор может выразить свою благодарность тем, кто вдохновил его на написание книги, кто помогал ему дельным советом или добрым участием, от кого он получал поддержку.

Не воспользоваться этой возможностью было бы верхом черствости. Автор признателен своей семье, в особенности великолепному Ламику, но, конечно, больше всех – крошке Ру.

Введение

Программирование богато алгоритмами и структурами данных, и современный программист должен владеть большинством из них. Или, по крайней мере, иметь о них ясное представление.

Да, нельзя объять необъятного. И все же есть некий минимум, которым, на наш взгляд, должен владеть всякий программист, чтобы считаться профессионалом.

В физике известен второй закон термодинамики. В самой примитивной формулировке он постулирует невозможность вечного двигателя: какие бы ухищрения нами ни предпринимались, невозможно направить всю имеющуюся энергию на полезную работу; часть (и порой значительная часть) энергии неизбежно будет теряться. Конечно, закона сохранения энергии никто не отменял: сама по себе энергия никуда не пропадает, она переходит в другие формы и рассеивается в виде тепла. Именно поэтому КПД любого двигателя всегда строго меньше единицы. Энергия, которой принципиально нельзя воспользоваться, увеличивает хаос во Вселенной. Количественной мерой этого хаоса (или неупорядоченности) в термодинамике служит энтропия. Задача инженеров – уменьшить энтропию, снизить, насколько можно, потери, не дать этой рассеянной энергии еще больше увеличить хаос.

Программирование – это дисциплина на стыке математики и инженерии. Как математики мы, программисты, призваны решать сложные задачи, переводить их на язык чисел и символов. Как инженеры – воплощать решения в форме программ, которые могут быть выполнены машиной. Кроме того, мы должны обеспечивать эффективность выполнения этих программ. Эффективность – это не только скорость, с которой задача будет решена, но также и объем занимаемой памяти. Разумеется, к инженерным «заботам» следует отнести масштабируемость, расширяемость, устойчивость, возможность внесения изменений и проч.

Математик работает так, как будто у него в запасе вечность и неограниченные ресурсы. Если для решения задачи программе придется проработать пусть даже миллион лет, математик будет считать свою миссию выполненной. Инженер, напротив, скован массой технических и экономических ограничений. Миллион лет его никак не устраивает:

хорошо – час, может быть – день, в крайнем случае – месяц. Если инженер не может обеспечить этого, то либо задача признается (в данных условиях) нерешенной, после чего она возвращается математику для дальнейших раздумий, либо возникают сомнения, и часто обоснованные, в том, что инженер вообще способен это сделать.

Программист – это, как правило, немного математик, а в основном инженер (многие, очень многие из тех, кто считает себя программистами, совсем не понимают самой необходимой математики; может, это и чересчур, но вряд ли имеют право так себя именовать). И большую часть времени программист занимается второй, т. е. инженерной, стороной деятельности.

Без надежных и удобных инструментов любая деятельность, а инженерная – в особенности, неэффективна и даже невозможна. Математика снабжает нас идеями, концепциями и рецептами (в программировании обычно говорят об алгоритмах), но именно инструментарий делает нас сильными, позволяет воплощать идеи, концепции и рецепты в работающие программы. У хорошего мастера, в какой бы области он не трудился, есть свой набор инструментов: у любого плотника мы найдем пилу, рубанок и стамеску; электрик не может обойтись без отвертки, плоскогубцев и изоленты, а строителю необходимы лопата, мастерок и рулетка. А всем им необходим молоток.

Программисту тоже необходимы инструменты. Понятно, первый из них – компьютер. Но без программ компьютер бесполезен (разве что обогревать пространство вокруг себя, увеличивая энтропию Вселенной). А программы – это не только машинные коды, способные выполняться процессором, но и данные, обычно организованные особым образом. Если программист действительно хочет быть профессионалом, а не поденщиком, он должен иметь ясное представление о способах представления данных, понимать, когда их применять, и уметь ими пользоваться.

Одной из таких структур данных – стеку – и посвящена эта небольшая книга. Стек давно известен и заслуженно пользуется уважением; без него сегодня немыслимо написать ни одной программы. Порой стек используется явно, но куда чаще он действует скрытно, в глубине. И надо признать, именно эта – неочевидная – часть возможностей остается для многих программистов terra incognita. Очень жаль...

Программисты в своей ежедневной работе полагаются на операционные системы, средства разработки, языки программирования. Это – часть их инструментария, в котором также используется стек. Конечно, можно было бы сказать: «Для чего мне знать что-то еще? До-

статочно того, что это работает и этому можно доверять». Да, такой взгляд имеет право на существование. До тех пор, пока мы решаем типовые задачи (а для большинства такие задачи – единственное, чем им приходится заниматься всю профессиональную карьеру), большего и не нужно. Но программирование неизмеримо богаче. В нем полным-полно задач нерешенных, решенных частично либо решенных плохо и ждущих своего часа. Стоит ли сознательно обеднять себя и обманываться тем, что можно обойтись имеющимся? Или ждать, когда кто-то более умный и отважный сделает то, что считалось невыполнимым?

Конечно, каждый решает этот вопрос по-своему. Но отвергнуть, даже не попробовав, никуда не годится.

Изложение в книге построено следующим образом. Стек не возникает неизвестно как, откуда и для чего. В книге рассматривается ряд задач нарастающей сложности. Все эти задачи рано или поздно приводят к стеку; таким образом, всякий раз появление стека подготавливается и обосновывается. Для того чтобы читатель мог составить себе наиболее полное представление о стеке и его возможностях, часто приводится решение исходной задачи другими способами. Такое сравнение всегда полезно, поскольку позволяет оценить достоинства и недостатки различных подходов и выбрать из них лучший.

Книгу лучше читать по порядку, поскольку последующий материал обычно опирается на предыдущий. Там, где это представлялось полезным, более ранний материал частично повторяется; это избавляет от необходимости возвращаться назад и позволяет поддерживать определенный ритм.

Структурно книга состоит из трех частей. Первая часть – основная. Именно здесь ставятся, обсуждаются и решаются задачи. Вторая часть посвящена описанию небольшой, но достаточно мощной, стековой машины и программированию для нее. Третья часть – это приложения, среди которых – исходные коды транслятора и интерпретатора стековой машины, описанной во второй части.

Завершается книга библиографией. Мы приложили все усилия к тому, чтобы перечислить в ней действительно ценные (на наш взгляд) и в то же время доступные источники, относящиеся к основной теме книги.

Эти источники позволят заинтересованным читателям двигаться дальше – к настоящим высотам.

В ежедневной практике программиста стек в явном виде встречается лишь иногда, большей частью стек «трудится» незаметно. Чаще всего он напоминает о себе тогда, когда работа программы внезапно

прекращается и на экран (либо в консоль) выводятся малопонятные непосвященному строчки сообщений, состоящие из адресов памяти и имен функций, вызовы которых привели к остановке. Как правило, такие сообщения свидетельствуют о том, что в программах имеются ошибки. Эти ошибки обычно невозможно отследить на этапе трансляции программы; они проявляют себя во время исполнения. Многие (возможно, большинство) не понимают ни причин их возникновения, ни того, что они означают. Прочитав книгу, вы начнете понимать, отчего это происходит и что нужно делать, чтобы программы стали более надежными и устойчивыми.

Основная область применения стеков – трансляция языков программирования и поддержка сред исполнения. Эти области информатики считаются сложными, и это действительно так. Но даже самые сложные вещи состоят из простых. Вот этими, относительно простыми вещами мы займемся и коснемся многого, что непосредственно используется при реализации языков программирования и работы программ после их запуска на исполнение. Если когда-нибудь вам придется (или захочется) реализовать язык программирования, то будьте уверены – стек станет вашим главным помощником.

И последнее. Стек в качестве главного «героя» был выбран не случайно. Он по-своему красив. Правда, для того чтобы это оценить, придется потрудиться. Нельзя понять музыку, читая рецензии; музыку надо слушать. То же и со стеком – с ним надо поработать. В основном тексте книги почти нет привычных задач и упражнений; упражнением в известном смысле является весь текст. Если читатель попробует свои силы в реализации задач, которые рассматриваются в книге, – это лучшее, чего можно пожелать. Польза будет несомненной.

Кое-где в тексте в качестве иллюстрации тех или иных понятий встречаются фрагменты программ, написанных на языке программирования Java. Все они очень просты, и даже если читатель незнаком с Java, мы уверены, что он без труда их поймет.

Мы нигде не используем примечаний: ни подстрочных, ни вынесенных в отдельный раздел. Многие склонны их рассматривать как малозначительные отступления и обычно пропускают. Кроме того, такие примечания прерывают процесс чтения и вынуждают отвлекаться от основной темы. Мы сделали примечания частью текста, *выделяя их курсивом*. Не стоит их игнорировать – в них много полезной информации.

Часть I

Задачи, приводящие к стеку



Скобочные структуры: элементарный случай

Мы начнем с того, что рассмотрим одну простую задачу, которая послужит отправной точкой всего последующего изложения. Эта задача встречается в разнообразных вариантах (и чуть позже мы расскажем об этом подробнее), но нам пока вполне достаточно такой формулировки: определить – правильно ли расставлены скобки в произвольном алгебраическом выражении. Несколько примеров даст представление о том, что мы имеем в виду и какое решение ищем:

```
a + (b * c /(d - e))
a + (b * c /d - e)
(a + b - c * (d / (e + f)))
```

Если всем входящим в эти выражения переменным (a, b, ...) придать определенные числовые значения, то эти выражения могут быть вычислены по правилам элементарной арифметики (конечно, исключая случаи, когда знаменатели дробей обратятся в 0). В этих примерах выражений скобки расставлены правильно. Чтобы сделать последнее утверждение более наглядным, давайте оставим только скобки, убрав все «лишнее»; в результате у нас получится следующее:

```
(())
()
((()))
```

Такого рода выражения будем называть *скобочными структурами*; мы полностью игнорируем все то, что содержится внутри скобок, и сосредоточиваем внимание только на скобках и их взаимном расположении. Все приведенные выше скобочные структуры, очевидно, корректны, т. е. сформированы правильно. Теперь приведем несколько примеров некорректных (неправильно сформированных) скобочных структур:

```
(()
())
)(
```

Давайте немного остановимся на последних примерах и проанализируем, что именно делает эти скобочные структуры некорректными (ошибочными или неправильно сформированными). Для этого удобно называть левые скобки «(» открывающими, а правые «)» – закрывающими.

В первом примере число открывающих скобок больше числа закрывающих. Во втором примере ситуация противоположная – число открывающих скобок меньше числа закрывающих. В таких случаях говорят, что скобки *не сбалансированы* по числу вхождений. Третий пример демонстрирует еще один случай: число открывающих скобок совпадает с числом закрывающих (т. е. скобки сбалансированы по числу вхождений), но скобочная структура тем не менее ошибочна – выражение не может начинаться с закрывающей скобки или заканчиваться открывающей.

Наша ближайшая задача – найти способ (алгоритм) проверки любой заданной скобочной структуры, состоящей только из скобок одного типа (в данном случае скобочной структуры, состоящей только из круглых скобок).

Очевидно, что существует бесконечное количество как правильно, так и неправильно сформированных скобочных структур. Поэтому попытка описать все возможные варианты скобочных структур их простым перечислением не осуществима даже теоретически – рано или поздно встретится такая скобочная структура, которая не была нами предусмотрена. В таком случае программа, осуществляющая проверку скобочной структуры, будет не в состоянии определить ее правильность. Следовательно, нужен иной способ. Такой способ (и очень простой) существует, но, перед тем как описывать его, мы предлагаем читателю немного подумать и попытаться найти его самостоятельно.

Если последовательность скобок начинается с закрывающей или завершается открывающей скобкой (наш третий пример), то, бесспорно, такая скобочная структура сформирована неправильно. Но эти два предельных случая не позволяют решить задачу в общем виде, т. к. первые примеры неправильно сформированных скобочных структур начинаются и завершаются нужными видами скобок – соответственно «(» и «)», но которые тем не менее ошибочны.

Итак, в решении задачи мы пока почти никуда не продвинулись. Но давайте посмотрим на проблему чуть иначе. Отвлечемся на несколько минут от скобок. Представьте лифт. Во избежание аварии необходимо контролировать массу груза (включая людей), перевозимого лифтом. Для этого в лифт встраивается специальный датчик. Каждый внесенный груз или входящий человек увеличивает общую массу груза в кабине и нагрузку на тросы и лебедку лифта. Разумеется, показания датчика увеличиваются. Каждый выходящий из кабины лифта уменьшает массу груза в кабине, и показания датчика уменьшаются.

Наш «груз» – это два вида скобок. Открывающая скобка увеличивает «массу», а закрывающая – уменьшает. Конечно, как и всякая аналогия, наш пример страдает неточностями и упрощениями, но он тем не менее наводит на идею.

Припишем каждой скобке некий «вес»: открывающей скобке – положительное число (скажем, 1), а закрывающей – число, равное по абсолютной величине, но противоположное по знаку (т. е. –1). Датчиком пусть будет целочисленная переменная (назовем ее, например, count) с начальным значением, равным 0 (в примере с лифтом это означает, что изначально кабина пуста). Эта переменная будет играть роль счетчика. Промоделируем поведение нашей «системы» на самой простой скобочной структуре: `()`. Для компактности будем представлять отдельные состояния в виде таблицы, в левой колонке которой указывается уже прочитанная часть скобочной структуры, а в правой (в той же строке) – значение счетчика. Получаем следующий протокол разбора:

```
Прочитанная часть   Счетчик
                       0
(                      1
()                     0
```

В самом начале (когда еще ни одна скобка не прочитана) счетчик равен 0. После обнаружения открывающей скобки соответствующее

ей число (т. е. 1) прибавляется к счетчику. После обнаружения закрывающей скобки соответствующее ей число (т. е. –1) также прибавляется к счетчику. Ясно, что итогом арифметической операции 1 + (–1) является 0 (лифт пуст). Если скобочная структура сформирована правильно, то значение счетчика по окончании разбора будет равно 0. Для закрепления (и дополнительной проверки) разберем другую скобочную структуру `(())`. Вот протокол разбора:

```
Прочитанная часть    Счетчик
                       0
(                      1
((                     2
(()                    1
(())                   0
```

Похоже, что для правильно сформированных скобочных структур этот метод работает. А что можно сказать о неправильно сформированных? Рассмотрим это на примере скобочной структуры `(()`:

```
Прочитанная часть    Счетчик
                       0
(                      1
((                     2
(()                    1
```

Вся скобочная структура прочитана, а значение счетчика положительно (в нашем случае равно 1). Если обратиться к аналогии с лифтом, то это означает, что кто-то вошел в лифт и не выходит из него. Из этого немедленно следует, что число открывающих скобок в скобочном выражении больше числа закрывающих и скобочное выражение в целом сформировано неверно. Противоположный случай `())` дает следующий результат:

```
Прочитанная часть    Счетчик
                       0
(                      1
()                     0
())                   -1
```

Вся скобочная структура прочитана, а значение счетчика отрицательно (в нашем случае равно –1). Из этого немедленно следует, что число открывающих скобок в скобочном выражении меньше числа закрывающих и скобочное выражение сформировано неверно (из лифта вышло больше людей, чем в него вошло, что, конечно, невозможно). Наконец, рассмотрим случай `)(`:

```
Прочитанная часть    Счетчик
                        0
)                      -1
```

Как только значение счетчика станет отрицательным, то оставшуюся часть скобочной структуры можно не проверять (если лифт пуст, то из него некому выходить) – она заведомо ошибочна, и, следовательно, вся эта скобочная структура сформирована неверно. Необходимо сразу прекратить разбор и отвергнуть эту скобочную структуру как недопустимую. Почему нужно прекратить разбор, не проверяя следующих скобок? Во-первых, это бессмысленно, а во-вторых, открывающая скобка установит значение счетчика в 0, и мы можем решить, что такая скобочная структура допустима, хотя, очевидно, это не так. В качестве легкого упражнения проведите разбор следующей скобочной структуры: (())). Первые четыре скобки установят значение счетчика в 0, но последняя (закрывающая скобка) лишняя, и все скобочное выражение должно быть отвергнуто. Кстати, продолжая, что можно сказать о скобочных структурах (()()), (())() или (()()?

Итак, задача решена. Для скобок одного типа (в нашем примере круглых) существует простой и эффективный алгоритм проверки структуры скобок.

Тип скобок может быть и иным, например фигурные {}, угловые <> или квадратные []. Более того, помимо указанных скобок, в языках программирования используются и другие конструкции, которые могут быть отнесены к скобкам, например begin-end или try-catch (правда, обычно такого рода конструкции называют блоками, но суть дела от этого принципиально не меняется). По-существу, скобками являются и многострочные комментарии вида /* ... */. Иными словами, не важно, что мы называем скобками и как они выглядят; скобкой может быть все, что выполняет функции скобки.

Последнее утверждение служит иллюстрацией т. н. «утиного теста»: если нечто похоже на утку, плавает как утка и крякает как утка, то это, скорее всего, утка. Иначе говоря, нет необходимости связывать себя только общепринятыми типами скобок; важна сущность, выполняемые функции.

Простые скобочные структуры, подобные рассмотренным только что, составляют лишь небольшую часть синтаксических структур в языках программирования. Обычно встречаются более сложные скобочные структуры, т. е. скобочные структуры, включающие разные типы скобок. Вот небольшой пример:

```
{
int [] keys;
ArrayList <String> aList;
...
for (int i = 0; i < keys.length(); i++) {
        aList.get (i) = "";
    }
}
```

Перед нами фрагмент исходного кода (на языке программирования Java, но подобный пример можно встретить во многих других языках), в котором встречаются все четыре «стандартных» типа скобок. Задача все та же: определить, правильно ли сформирована скобочная структура в том случае, когда типов скобок не один, а несколько. Давайте, как и в первом случае, уберем все «лишнее» и оставим только скобки. Вот что у нас получится:

```
{[]<>(()){()}}
```

Как убедиться в правильности скобочной структуры такого вида? Сразу же приходит в голову мысль воспользоваться только что разработанным методом подсчета, приписывая каждой открывающей и каждой закрывающей скобке некий «вес», т. е. определенное число, характеризующее вид скобки – открывающая или закрывающая. Пусть, как и прежде, любой открывающей скобке соответствует 1, а любой закрывающей –1. Для нашего примера после просмотра скобок получится 0 (и при этом счетчик никогда не станет отрицательным), т. е. мы можем сделать вывод, что скобочная структура правильна.

Итак, задача решена? Как бы не так! Посмотрим на такую, например, скобочную структуру: {(}). Наш алгоритм даст в результате значение 0, при этом значение счетчика никогда не будет отрицательным. Но эта скобочная структура, совершенно очевидно, сформирована неверно. Вывод: алгоритм со счетчиком не работает. Хуже всего то, что он выдает результат, который выглядит правильным (т. к. счетчик по окончании разбора действительно равен 0), но которому нельзя верить. Надо понять – почему он не работает и можно ли его исправить так, чтобы он был пригоден в новой ситуации с различными типами скобок.

Вообще говоря, причина лежит на поверхности: мы увеличиваем (соответственно, уменьшаем) значение счетчика всякий раз, когда в последовательности встречается открывающая (закрывающая)

скобка. В примере {(}) мы увеличиваем значение счетчика последовательно дважды: сначала, когда встречается скобка {, и потом, когда встречается скобка (. Но ведь это разные типы скобок! Не тут ли корень проблемы? Нельзя ли изменить способ изменения значений счетчика так, чтобы он соответствовал типу скобки? Можно, конечно, и будет правильно, если читатель попробует поискать решение в этом направлении.

Методы анализа скобочных структур стали исследоваться еще в 50-х годах XX века. Если в выражении встречаются скобки только одного вида, то задача, как мы уже видели, решается элементарно. Но по мере усложнения и развития языков программирования скобочные структуры стали включать в себя скобки разных типов, и проблема стала актуальной. Был предложен ряд методов решения (в их числе весьма оригинальные), но все эти методы были сложными в реализации и медленными в работе. Если читатель уже попытался найти такой метод, он, безусловно, должен был убедиться, что проблема нетривиальна. Можно вводить отдельные счетчики для каждого вида скобок, но проблема согласования вложенности одних типов скобок в другие все равно оставалась. Метод со счетчиком, при всей его элегантности, для составных скобочных структур работает неудовлетворительно. Нужно что-то иное, и сейчас мы переходим к обсуждению метода, совершенно отличного от метода подсчета с использованием счетчика.

Вернемся еще раз к нашему примеру {(}). Он, как мы знаем, соответствует недопустимой скобочной структуре. Проанализируем скобки по порядку. Первой встречается открывающая { скобка. Чтобы скобочная структура стала допустимой, где-то в последовательности скобок должна быть закрывающая } скобка. Она, конечно, присутствует (в примере это третья скобка слева). Но, прежде чем мы до нее «доберемся», надо что-то сделать с другой открывающей скобкой, только теперь это (. Для того чтобы (третья по порядку) закрывающая фигурная скобка } соответствовала первой открывающей {, нужно обеспечить для открывающей круглой скобки парную ей закрывающую. Принято говорить, что круглая скобка (вложена в фигурную {, или, чуть более формально, что у открывающей круглой скобки *глубина вложенности* больше, чем у открывающей фигурной. Прежде чем закрывать внешние скобки (в данном случае { и }), нужно, чтобы были закрыты все внутренние. В нашем примере как раз это очевидное правило и не соблюдается: фигурная скобка }, относящаяся к паре с меньшей глубиной вложенности (внешней), появляется

раньше круглой), которая относится к скобкам большей глубины вложенности (внутренней).

Может быть, небольшой пример поможет лучше понять эти довольно абстрактные рассуждения. Представьте себе закрытую шкатулку, внутри которой лежит закрытый конверт. Нам нужно прочесть содержимое конверта, т. е. лежащее в нем письмо, а затем все вернуть в исходное состояние. Шкатулка соответствует скобкам { и }, а конверт – скобкам (и). Прежде чем мы доберемся до конверта, надо открыть шкатулку. Этому соответствует скобка {. Далее мы должны открыть конверт, т. е. скобку (. После этого мы достаем письмо, читаем его, прячем обратно в конверт и закрываем конверт; этому соответствует, разумеется, закрывающая) скобка. Наконец, мы закрываем шкатулку, используя скобку }. Результат, очевидно, такой: {()}, и это правильно сформированная скобочная структура с правильным порядком вложенности. Наш начальный пример {(}) соответствует ситуации, которую можно описать так: открыть шкатулку, открыть конверт, достать письмо, закрыть шкатулку, закрыть конверт. Но как раз последнее действие выполнить и невозможно – шкатулка закрыта, и до конверта, лежащего в ней, добраться уже невозможно!

Работая над этой задачей, мы после некоторого размышления рано или поздно приходим к выводу о том, что решение, основанное на использовании счетчика (или даже нескольких счетчиков), по-видимому, следует отбросить. Счетчик хорошо работал для первого случая, когда у нас были скобки только одного типа. Его было вполне достаточно для хранения всей информации об уже просмотренном участке скобочной структуры; последующие скобки только модифицировали состояние счетчика, но не меняли существа обрабатываемой информации – тип скобки оставался прежним, и мы могли его игнорировать. Как только типов скобок стало больше, метод перестал работать, он подошел к границам применимости. Поступающая информация уже не может быть сохранена только с использованием счетчиков – информации не только стало больше, сама информация стала другой.

Метод с использованием счетчика прост и понятен: нам привычно что-то подсчитывать. Но всему наступает предел, за который надо выходить.

А теперь, после этого несколько затянувшегося описания причин безуспешных попыток анализа скобочной структуры с различными типами скобок, мы переходим к главному действующему лицу – стеку. Но, прежде чем мы продолжим, нам придется ненадолго прерваться.

Следующие несколько страниц будут очень важными, поскольку мы собираемся рассказать, что такое стек, опишем относящуюся к стеку терминологию, введем обозначения, которые позволяют избавиться от многословных описаний, и, наконец, расскажем о некоторых основных операциях со стеком. Возможно, не сразу будет понятно – для чего мы на время отложили нашу задачу, но потерпите. Все это необходимо будет тщательно изучить, так что давайте приступим.

Стек: знакомство

Обратимся к классике и позаимствуем определение стека у Дональда Кнута:

Стек – это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются только на одном из концов списка.

Для знатока структур данных этого, скорее всего, будет достаточно. Но все-таки давайте уделим определению стека некоторое время. Самое важное в этом определении – последние слова, а именно «только на одном из концов». Наверное, проще всего проиллюстрировать, что такое стек, – привести несколько примеров, которые всем, надеемся, будут знакомы и понятны.

Первый пример стека мы обнаруживаем в детской игрушке. Всем, вероятно, знакома детская пирамидка: закрепленный на основании стержень, на который надеты разноцветные диски. Для того чтобы поместить на стержень новый диск, необходимо надеть его на свободный конец стержня. Чтобы убрать со стержня диск, его необходимо снять со свободного конца стержня. Таким образом, и добавление, и удаление дисков осуществляются всегда только с одного конца стержня; ни снимать, ни надевать диски со стороны основания пирамидки нельзя. Далее, невозможно снять диск, над которым есть другие диски, и невозможно добавить новый диск иначе, как поверх других. Диск, добавленный последним, можно будет снять первым; диск, добавленный первым, будет снят в последнюю очередь.

Еще один классический пример стека – стопка подносов в кафе самообслуживания. Мы можем взять только самый верхний поднос из стопки. Когда подносы заканчиваются, сотрудник кафе приносит новые. Поднос на самом верху стопки – это последний добавленный поднос; поднос на дне стопки – самый первый добавленный поднос.

Нет возможности (если, конечно, не стоит задача все поломать и испачкать) взять поднос из середины стопки. Равно как и нет возможности добавить поднос в середину стопки.

Чтобы, наконец, перейти к делу, приведем еще один пример: магазин для хранения патронов в огнестрельном оружии. Очевидно, что первым будет использован тот патрон, который был помещен в магазин последним. Патрон, добавленный в магазин первым, будет использован последним.

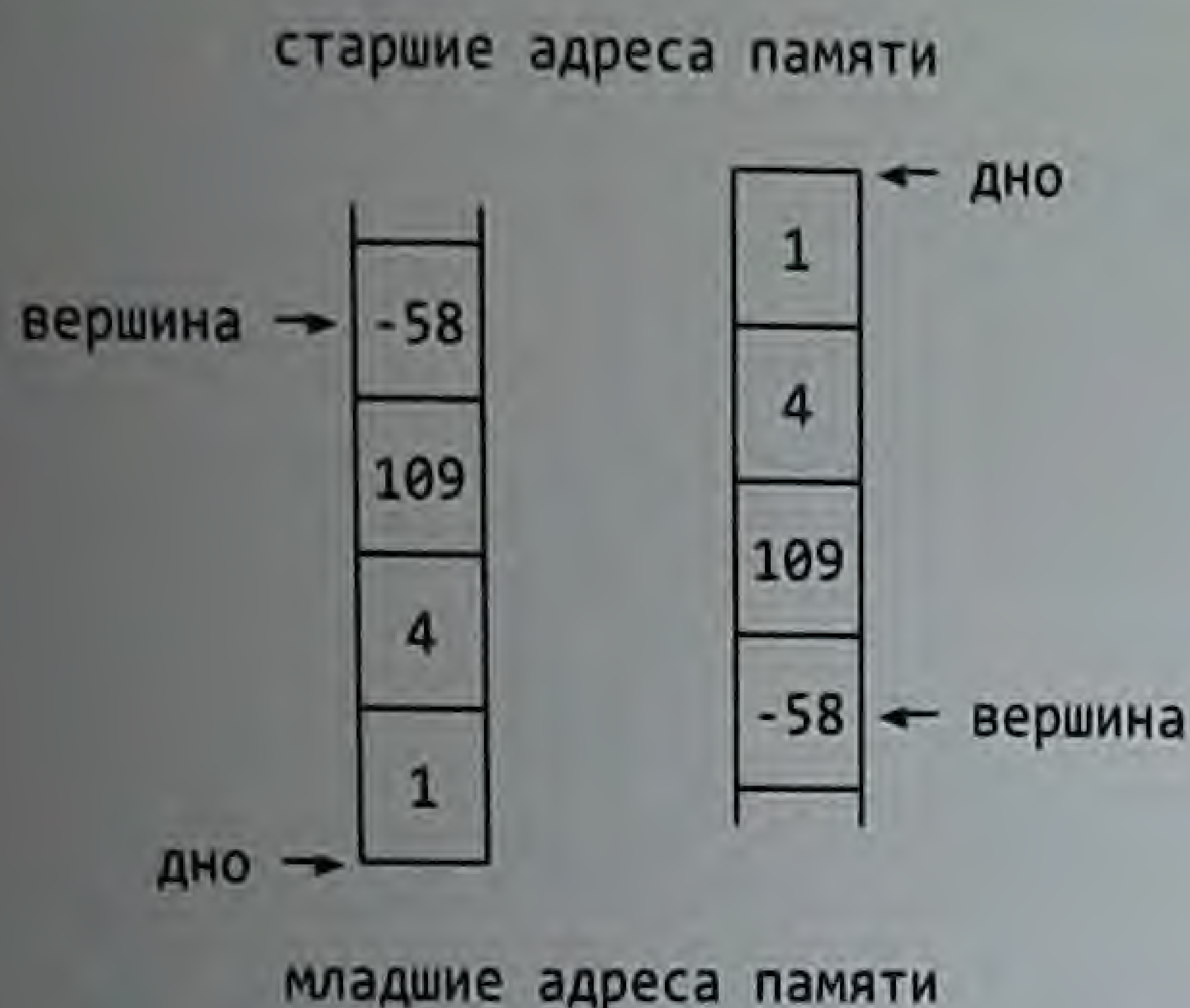
Во всех трех примерах мы видели один и тот же тип поведения: то, что было добавлено первым (диск, поднос, патрон), будет использовано последним. То же самое можно сказать и так: то, что было добавлено последним, будет использовано первым. Как раз для этой формы определения существует эквивалентная краткая *«Last In First Out»*, или просто – *LIFO*.

Итак, операции вставки и удаления из определения, приведенного в начале раздела, подчиняются дисциплине LIFO. Такая структура данных и есть стек.

Теперь, когда общая идея стека стала (надеемся) понятной, настало время договориться о том, как стек можно изображать графически. Существуют три основных способа изображения стека, и все они широко используются в литературе по программированию и структурам данных. Первые два способа практически идентичны друг другу. Обсудим сначала их.

Поскольку стек, как и всякая другая структура данных (такая как массив, список, дерево и проч.), хранится в памяти, в основе первых двух способов представления стека лежит отображение стека как набора ячеек памяти. Напомним, что память – это последовательность ячеек. Каждая ячейка имеет номер (от 0 до некоторой величины), или *адрес*. Ячейки памяти с меньшими адресами считаются расположенными ближе к началу, а с большими – ближе к концу памяти. Все ячейки памяти совершенно равноправны, и нет никаких оснований считать те или иные из них более или менее важными.

Графически память изображается в виде последовательности смежных ячеек. Стек может расти как в направлении от меньших адресов к большим, так и наоборот – от больших адресов к меньшим. Если условиться, что ячейки с меньшими адресами (т. е. те, что расположены ближе к началу памяти) изображаются ниже ячеек с большими адресами, то первые два способа изображают стек так, как показано на следующем рисунке:



На левом рисунке стек растет в направлении от меньших адресов к большим (т. е. снизу вверх), на втором – от больших адресов к меньшим (т. е. сверху вниз). По поводу изображенных на рисунках стрелок с надписями «дно» и «вершина» мы вскоре поговорим подробнее.

В большинстве компьютерных архитектур стек, как правило, растет сверху вниз, т. е. от больших адресов к меньшим (правый рисунок, см. выше). В дальнейшем мы будем придерживаться именно такого способа изображения стека, но имейте в виду, что часто можно встретить и другой способ (тот, что слева). Выбор направления не принципиален, и программист вправе использовать тот, который кажется ему более подходящим.

Конечно, интуитивно левый вариант кажется более привлекательным, т. к. мы привыкли считать от меньших значений к большим, но повторимся, что направление счета – это вопрос соглашения.

Третий способ представления стека позволяет изображать стек не в виде рисунков, а в виде строки, которую можно читать привычным способом, т. е. слева направо. Он, конечно, несколько менее нагляден, чем рисунки, но у него имеется одно бесспорное преимущество – компактность. Этот способ представления стека выглядит следующим образом:

```
├ 1 4 109 -58
```

Здесь изображен стек с тем же самым содержимым, что и в первых двух случаях. Стек растет слева направо. Дно стека обозначается символом «├» (впрочем, способ обозначения непринципиален, и мы могли бы выбрать любой другой символ или вовсе ничего не исполь-

зовать). Вершина стека обычно никак не обозначается; считается, что вершина – это просто самый правый элемент в строке.

Преимущество последнего обозначения не только (и даже не столько) в его компактности, сколько в том, что он позволяет легко изображать изменения, происходящие в стеке, т. е. содержимое стека в различные моменты времени: ведь записывать в строку куда проще, чем рисовать. Допустим, мы провели над содержимым стека какие-то операции (об операциях речь пойдет чуть позже). Первые два способа изображения стека, конечно, наглядны и удобны, но трудоемки. Кроме того, их сложно сделать частью документации программного кода. Третий способ решает эти проблемы элементарно:

```
├ 1 4 109 -58 → ├ 1 4 109 -58 49
```

Здесь показаны два состояния стека: до операции (слева от стрелки) и после операции (справа от стрелки). Очевидно, что такая линейная форма записи состояния стека позволяет записывать (и документировать) все основные операции над стеком. Первые два способа для этого чересчур громоздки.

Способ записи состояния стека в виде строки называется *стековой диаграммой* (или *диаграммой стека*) и является общепринятым. Мы будем применять как графическое изображение стека в виде вертикальных ячеек памяти (при этом будет считаться, что стек растет сверху вниз – от больших адресов памяти к меньшим, что соответствует правому рисунку, приведенному выше), так и стековые диаграммы, отдавая последним предпочтение.

Теперь, когда мы обсудили основные способы изображения стеков, можно, наконец, обратиться к понятиям дна и вершины стека, которые, скорее всего, уже и без того вполне понятны.

Неформально говоря, *дно* – это место, начиная с которого накапливаются элементы стека. В терминах адресов памяти дно – это адрес ячейки памяти, с которого начинается участок памяти, выделенной стеку.

Если стек растет снизу вверх, то его дно – ячейка памяти с наименьшим выделенным адресом (не обязательно 0); если стек растет сверху вниз, то его дно – ячейка памяти с наибольшим выделенным адресом (не обязательно последним). Для лучшего понимания рассмотрим пример. Пусть для стека выделен участок памяти с адресами от 1000 до 2000 (включительно). Если стек растет снизу вверх, то дно стека располагается в ячейке с адресом 1000. Новые элементы последовательно «попадают» в ячейки памяти с адресами 1000, 1001,

1002, ..., 2000. Если стек растет сверху вниз, то дно стека располагается в ячейке памяти с адресом 2000, и теперь новые элементы последовательно «попадают» в ячейки памяти с адресами 2000, 1999, 1998, ..., 1000.

Вершина стека – это ячейка памяти, в которую был добавлен или из которой будет удален последний добавленный элемент стека. Рассмотрим пример:

```
├ → ├ 1 → ├ 1 99 → ├ 1 99 -13
```

Пусть изначально стек был пуст (т. е. не содержал никаких элементов). Затем мы добавили в стек число 1. На него указывает *вершина стека*. Теперь последовательно добавим в стек число 99 и число –13 (см. рисунок). Очевидно, что стек содержит уже три элемента (в порядке добавления 1, 99, –13), т. е. стек не пуст. На вершине стека – последнее добавленное число (–13). Продолжим и удалим из стека один элемент. Так как стек – это структура данных, подчиняющаяся дисциплине LIFO, то удаляемым элементом будет последний добавленный (т. е. –13), число 99 теперь будет на вершине стека, и стек примет следующий вид:

```
├ 1 99
```

Вместо термина «вершина» программисты часто используют термин «голова» стека. Вместо термина «дно» встречается термин «основание» стека. Операция добавления нового элемента в стек обычно называется «проталкиванием» в стек (вспомните аналогию с магазином огнестрельного оружия). Операция удаления элемента часто называется «выталкиванием» из стека. Мы будем широко использовать все эти термины, т. к. они довольно точно отображают основные понятия стека и операций над ним.

В программах и описаниях алгоритмов для обозначения дна и вершины стека часто используются обозначения: соответственно bottom (или просто bot) и top (или иногда tail и head соответственно).

Стек предназначен для хранения в нем любой информации – не только чисел, но и символов и даже структур. В нашей книге стек будет использоваться преимущественно для хранения чисел.

Покажем небольшой пример использования стека: переворот строки. Например, если на входе имеется строка HelloWorld, то необходимо на выходе получить строку dlroWolleH. Используя стек, сделать это проще простого:

- поочередно, символ за символом, протолкнуть в стек все символы, составляющие входную строку (т. е. протолкнуть H, затем e, потом l и т. д. вплоть до d);
- поочередно, символ за символом, вытолкнуть из стека все находящиеся в нем символы (сначала d, затем l и т. д. вплоть до W).

Ну разве не просто?

Теперь нам осталось обсудить *основные операции* над стеком. Пока нам понадобятся только три из них (мы будем вводить новые операции постепенно, по мере необходимости). С двумя из них мы уже познакомились. Первая операция – добавление (проталкивание) нового элемента в стек: *push*. Вторая операция – удаление (выталкивание) элемента из стека: *pop*.

Третья операция проверяет стек на пустоту: *empty*.

Кажется, что эти операции не нуждаются в особых комментариях, но давайте посмотрим, так ли это.

Что, если мы попытаемся удалить элемент из пустого стека? Это, бесспорно, ошибка, и именно для предотвращения этой ошибки (она называется *underflow* – опустошение, исчерпание) и вводится операция empty. Пока все логично: если операция empty вернет логическое значение true (т. е. истина), то использовать операцию удаления (выталкивания) элемента pop нельзя; в противном случае операция pop будет выполнена успешно. Теперь представим, что объем памяти, выделенный стеку, исчерпан и стек полностью заполнен. Это означает, что все ячейки стека заняты и адрес вершины стека достиг предельного значения (в примере стека, растущего от адреса 2000 до адреса 1000, предельное значение соответствует адресу 1000). Попытка добавить в стек новый элемент операцией push приведет к ошибке переполнения стека (*overflow*). Если среда исполнения позволяет контролировать переполнение стека (что, как правило, и бывает), то работа программы, пытающейся выполнить такую операцию push, обычно завершается некорректно. Если среда исполнения, напротив, не контролирует переполнения стека, то новый элемент в стек будет добавлен (в нашем примере по адресу 999), но это гораздо хуже, чем ошибка в первом случае. Тогда мы хотя бы знали, что ошибка случилась, программа завершилась досрочно и мы должны придумать другое решение и устранить причину возникновения ошибки (например, слишком большая глубина рекурсивных вызовов). Во втором случае мы фактически портим содержимое памяти, которая не принадлежит стеку. Может случиться, что такая программа даже доработает до конца и нормально завершится. Но можно ли будет ве-

рить результатам работы такой программы? Конечно, нет. Подобная скрытая ошибка очень коварна, поскольку создает иллюзию того, что все идет хорошо. Так что ситуация переполнения должна обязательно контролироваться.

На этом сказанного о стеке и об основных операциях над стеком достаточно, и мы можем вернуться к нашей задаче – анализу скобочных структур, состоящих из различных типов скобок.

Скобочные структуры: общий случай

Теперь, наконец, мы практически готовы рассмотреть общий случай анализа скобочных структур, т. е. таких структур, в которых встречаются любые виды скобок и в любой комбинации. Для этого мы воспользуемся только что введенным понятием стека. Начнем с (правильной) скобочной структуры {()}. Хотя, как мы помним, эта структура правильно разбирается с использованием счетчика, в общем случае счетчик оказывается бессильным. Оставим попытки работать со счетчиком и посмотрим, чем нам может помочь стек.

Общий принцип действия будет таким: когда в последовательности встречается любая открывающая скобка (т. е. скобка (, [, { или <), то мы проталкиваем ее в стек. Когда в последовательности встречается любая закрывающая скобка (т. е. скобка),], } или >), мы действуем согласно следующему правилу, которое, для краткости, назовем «сопоставлением»: если тип закрывающей скобки совпадает с типом скобки на вершине стека (а в стеке, напоминаем, лежат только открывающие скобки), то мы выталкиваем из стека открывающую скобку и переходим к следующей скобке во входной последовательности. Если тип закрывающей скобки не совпадает с типом скобки на вершине стека, то разбор немедленно останавливается, и мы фиксируем ошибку.

Конечно, на словах это выглядит длинно и непонятно, так что давайте обратимся к нашему примеру. Вот так выглядит протокол разбора (последовательность шагов разбора скобочной структуры пронумерована для удобства ссылок):

```
Прочитанная часть   Стек
                    |                 (1)
{                   | {               (2)
{(                  | { (             (3)
{()                 | {               (4)
{()}                |                 (5)
```

Напомним, что символ ├ обозначает дно стека. Будем предполагать, что в начале разбора скобочной структуры стек пуст; очистку стека можно обеспечить разными способами, но здесь об этом мы говорить не будем. Изначально ни одна скобка не прочитана (шаг 1). Теперь мы считываем первую скобку; это – открывающая фигурная скобка. Поскольку скобка открывающая, она, согласно нашему алгоритму, проталкивается в стек (шаг 2). Читаем следующую скобку. Она, как и первая, – открывающая, хотя и другого типа (а именно – круглая). Ее тоже проталкиваем в стек (шаг 3). Теперь в стеке находятся две открывающие скобки.

Продолжаем считывать скобки из входной строки и встречаем закрывающую круглую скобку. Тут вступает в действие правило сопоставления. Первое, что мы должны сделать, – сопоставить тип только что прочитанной закрывающей скобки с типом скобки на вершине стека. Типы совпадают (обе скобки одного типа, а именно – круглые). Выталкиваем из стека скобку (очевидно, круглую открывающую); пока что считанная последовательность сформирована правильно (шаг 4). Наконец, считываем последнюю скобку. Это закрывающая скобка. Опять применяем правило сопоставления. Тип закрывающей скобки совпадает с типом скобки на вершине стека. Выталкиваем из стека открывающую скобку. А теперь – внимание. Скобочная структура полностью прочитана (иначе говоря – последовательность скобок закончилась), а стек – пуст. Вывод: скобочная структура {()} сформирована правильно. Вот и все. Согласитесь – элегантно. Давайте закрепим навык и рассмотрим более сложную структуру {([])<>}. Последовательно выпишем состояния стека:

```
Прочитанная часть    Стек
                     ├                    (1)
{                    ├ {                  (2)
{(                   ├ { (                (3)
{([                  ├ { ( [              (4)
{([]                 ├ { (                (5*)
{([])                ├ {                  (6*)
{([])<               ├ { <                (7)
{([])<>              ├ {                  (8*)
{([])<>}             ├                    (9*)
```

Номера строк, в которых происходит сопоставление текущей прочитанной закрывающей скобки со скобкой на вершине стека, помечены звездочкой (*). Очень важно тщательно изучить этот пример и убедиться в том, что работа стека понята правильно. Мы предлагаем

самостоятельно придумать несколько примеров правильно сформированных скобочных структур и проверить алгоритм их разбора путем тщательного выписывания состояний стека.

Если скобочная структура состоит только из скобок одного типа (то, что ранее мы назвали элементарным случаем), то это, очевидно, частный случай общего типа скобочных структур. Наш алгоритм к этому случаю, разумеется, вполне применим. Иначе говоря, о методе с использованием счетчика можно вовсе забыть и всегда использовать метод с использованием стека.

Итак, мы убедились, что стек позволяет просто и корректно распознавать правильно сформированные скобочные структуры. Теперь надо убедиться в том, что этот метод позволяет распознавать и неправильные скобочные структуры. Рассмотрим уже знакомый нам пример {(}):

```
Прочитанная часть   Стек
                    ├                  (1)
{                   ├ {                (2)
{(                  ├ { (              (3)
{(}                 ├ { (              (4*)
```

Давайте подробно рассмотрим этот протокол разбора. В первых строках происходят чтение открывающих скобок (фигурной и круглой) и их проталкивание в стек. Пока все идет как надо. Но вот мы читаем закрывающую скобку } (шаг 4). В игру вступает правило сопоставления: необходимо сравнить тип закрывающей скобки с типом скобки на вершине стека. И тут обнаруживается, что их типы не совпадают. Мы не можем вытолкнуть из стека открывающую скобку, поскольку для нее нет парной закрывающей того же типа. Дальше можно уже не анализировать – скобочная структура составлена неправильно, и мы фиксируем ошибку. Закрепим результат, для чего рассмотрим скобочную структуру ((()}):

```
Прочитанная часть   Стек
                    ├                  (1)
(                   ├ (                (2)
((                  ├ ( (              (3)
(((                 ├ ( ( (            (4)
((()                ├ ( (              (5*)
((()}               ├ ( (              (6*)
```

Как и раньше, номера строк, в которых происходит сопоставление текущей прочитанной закрывающей скобки со скобкой на вершине

стека, помечены звездочкой (*). Поначалу все идет прекрасно – мы проталкиваем в стек три открывающие скобки, встречаем закрывающую скобку (круглую), сопоставляем ее со скобкой на вершине стека и выталкиваем (шаг 5) из стека открывающую скобку. Но вот мы считываем закрывающую скобку другого типа (а именно фигурную). Попытка сопоставления этой скобки со скобкой на вершине стека терпит неудачу (шаг 6), дальнейший разбор прекращается, и мы фиксируем ошибку.

Еще один простой пример:)(. Тут все элементарно. Мы считываем закрывающую скобку и пытаемся сопоставить ее тип с типом скобки на вершине стека. Но ведь стек пуст и сопоставлять нечего! Результат: скобочная структура)(ошибочна. Напоследок рассмотрим еще одну ситуацию: (){. Вот протокол разбора этой скобочной структуры:

```
Прочитанная часть   Стек
                    ├                  (1)
(                   ├ (                (2)
()                  ├                  (3*)
(){                 ├ {                (4)
```

Первые две скобки образуют правильную (допустимую) часть скобочной структуры (шаги 1, 2 и 3). Появление следующей скобки (шаг 4) обрабатывается, как и прежде, – поскольку скобка открывающая, то она проталкивается в стек. И тут скобочная структура заканчивается – скобок в ней больше нет. Разумеется, это ошибка, и скобочная структура (){ считается сформированной неверно. Для закрепления полученных знаний мы рекомендуем составить несколько скобочных структур (как правильно, так и неправильно сформированных) и тщательно проверить их только что рассмотренным способом.

Пора подводить некоторые итоги. Вспомните, сколько усилий мы потратили на то, чтобы проверять скобочные структуры, состоящие из скобок только одного типа. Мы должны были ввести счетчик и корректировать его значение всякий раз, когда считывалась очередная скобка. Но этот метод оказался слишком «слаб», как только скобочная структура стала включать в себя другие типы скобок. В чем причина этой «слабости»? Мы уже упоминали об этом, но тогда это было только предположение. Теперь мы готовы дать ответ.

Что нового дал нам стек? Он дал возможность *хранения* уже прочитанных и обработанных данных. Счетчик, при всей его простоте и привлекательности, не «помнил» предыстории скобок. Счетчик фиксировал только факты появления открывающих и закрываю-

щих скобок, но не их типы. Стек позволяет запоминать предыдущую информацию, сопоставлять эту сохраненную информацию с новой информацией и на основе результатов сопоставления принимать решение о дальнейших действиях. Иными словами, стек работает как *память*. Конечно, как память специфического вида, ведь в стеке доступ осуществляется только в одном месте – в его вершине. Информация, лежащая ниже вершины стека (иными словами, информация, добавленная раньше), не доступна. Но этого оказалось вполне достаточно для решения задачи.

Стек позволяет упорядочить информацию во времени: данные, поступившие раньше других (они хранятся ближе ко дну стека), будут доступны после данных, поступивших позже других (они хранятся ближе к вершине стека). Повторимся: стек хранит данные в порядке их поступления. Именно эта особенность – хранение информации в зависимости от времени ее поступления – делает стек поистине незаменимым, в чем мы не раз убедимся в дальнейшем.

Не будем откладывать и обратимся к следующей задаче, в которой стек предстанет перед нами в новом качестве.

Подпрограммы: постановка задачи

Важным и очень полезным средством разделения программ на отдельные и относительно независимые составляющие являются функции (function), процедуры (procedure), методы (method) и подпрограммы (subroutine). Они представляют собой набор действий, снабженный именем. Подпрограммы можно рассматривать двояко: как средство управления структурой программы и придания программе модульности путем разделения ее на небольшие части и как расширение языка программирования новыми операциями, определенными программистом. Каждая функция, процедура, метод или подпрограмма может быть многократно вызвана из различных частей программы, в том числе из других функций, процедур, методов и подпрограмм. В большинстве современных языков программирования возможны также их рекурсивные вызовы; об этой полезной, интересной, но достаточно непростой возможности мы поговорим позже, после того как детально разберемся с обычными (т. е. нерекурсивными) вызовами.

Принципиальных отличий между функциями, процедурами, методами и подпрограммами нет, и поэтому в дальнейшем мы будем использовать в качестве их общего имени один и тот же, исторически первый термин – *подпрограммы*.

Те немногие отличия, о которых стоит упомянуть, касаются возможности вырабатывать возвращаемый результат. Традиционно считается, что функции включают в себя конструкцию, возвращающую некоторое значение (не обязательно примитивного типа). Методы, а также подпрограммы могут как возвращать, так и не возвращать никаких результатов. Наконец, процедуры, как правило, никаких результатов не возвращают. Эти отличия носят в основном синтаксический характер и для нас не принципиальны.

Подпрограммы в разных языках программирования оформляются в соответствии с синтаксисом этих языков и могут выглядеть по-разному. Это, однако, не существенно; в конце концов, несмотря на внешние (видимые) отличия подпрограмм, записанных на разных языках программирования, все они, в сущности, делают одно и то же. Чтобы сделать это утверждение наглядным, не зависящим от особенностей того или иного языка программирования, и одновременно придать нашему обсуждению конкретности и убедительности, мы поступим следующим образом: введем небольшой и простой язык ассемблера, на основе которого продемонстрируем основные концепции управления подпрограммами.

Тот, кто раньше не сталкивался с ассемблерами, но слышал о том, что это очень сложные и малопонятные языки программирования, зачастую склонен прекращать чтение и откладывать книгу в сторону. Это – заблуждение. Конечно, ассемблер может быть сложным и малопонятным. Но давайте не станем торопиться: можно запутать, усложнить и сделать невразумительным что угодно, используя даже самые ясные конструкции (вспомните, сколько людей не могут внятно рассказать даже о самых простых вещах, например объяснить, как из пункта А попасть в пункт В).

Конечно, программирование на ассемблере имеет свою специфику, поскольку ассемблеры «располагаются» всего лишь в шаге от программирования в машинных кодах (т. е. от программирования на языке последовательностей нулей и единиц, которые только и «понимает» процессор). Но сложность сама по себе не должна и не может служить препятствием. В конце концов, программирование – это решение задач, среди которых есть действительно очень сложные, но которые мы тем не менее решаем (или хотя бы пытаемся решить). Наш ассемблер будет совсем простым; может быть, чуть-чуть сложнее обычного калькулятора. Так что давайте отбросим сомнения, а просто приступим к делу. Мы увидим, что в ассемблере нет ничего недо-

ступного; более того, с его использованием многие вещи становятся гораздо яснее и понятнее.

В основных чертах наш ассемблер подобен другим ассемблерам (с тем приятным отличием, что он действительно предельно прост и не зависит от особенностей конкретной архитектуры процессора). Его система команд (т. е. инструкции, которые он «понимает» и способен выполнять), по сравнению с другими ассемблерами, невелика, что, конечно, делает его не слишком удобным для практического программирования, но при необходимости эту систему команд легко расширить. Собственно, именно так мы и поступим. Команды будут вводиться постепенно и только тогда, когда это будет нужно. Таким образом, необходимость введения новых команд будет всякий раз обосновываться, а новые команды будут появляться лишь тогда, когда целесообразность этого станет очевидной и необходимой.

Для начала введем несколько терминов, которые будут нами в дальнейшем широко использоваться. *Вызов* подпрограммы означает, что следующими должны будут исполняться команды, составляющие *тело* подпрограммы. Другое название для этого действия – *передача управления* на начало подпрограммы, или, что то же самое, *передача управления по адресу начала* подпрограммы, т. е. по адресу памяти, начиная с которого расположена подпрограмма. Ну а адрес, как мы помним, – это просто номер (неотрицательное целое число) ячейки памяти.

Теперь следует разобраться, как мы можем задать или указать адрес. Для этого вводится понятие *программного счетчика* (или, как иногда говорят, *счетчика размещения*), который будет обозначаться как PC (от program counter – не путать с personal computer). Программный счетчик – это регистр, обычно входящий в состав процессора. На некоторых архитектурах PC располагается не внутри процессора, а на т. н. общей шине, и в этом случае он, как и другие ячейки памяти, имеет свой адрес. Для наших целей то, где фактически располагается PC, значения не имеет. PC содержит (хранит) адрес команды, которая будет выполнена следующей. Арифметико-логическое устройство (АЛУ) процессора выполняет команду, находящуюся по текущему адресу, а затем передает управление по адресу, который находится в PC. После этого содержимое PC корректируется так, чтобы он указывал на следующую команду. Тем самым обеспечивается непрерывный цикл выборки команды, исполнения команды и перехода к адресу следующей команды. Теперь обратимся к командам.

На первых порах нам потребуется только команда *безусловного перехода* BR (BR – это *мнемоника*, или сокращение от branch, т. е. «ветка»). Встретив эту команду, управление передается по адресу, содержащемуся в PC. Иначе говоря, команда BR соответствует оператору goto в некоторых языках программирования (например, Fortran, C, Pascal, некоторых версиях Basic'a). Команда BR прерывает линейную последовательность выполнения команд и передает управление команде, расположенной в произвольной ячейке памяти. Следующий небольшой пример продемонстрирует, что при этом происходит. Допустим, что PC содержит адрес 2000. Удобно использовать обозначение [PC]=2000, где квадратные скобки читаются как «содержимое чего-либо», т. е. в нашем случае «содержимое PC» (в литературе и документации часто используются также обозначения (PC) и c(PC) от слова content). Это означает, что следующей будет выполнена команда, которая хранится по адресу 2000. Это т. н. *эффективный адрес*, но нам данный термин не понадобится. Пусть по адресу 2000 хранится команда BR 2500 (т. е. [2000]=BR 2500):

```
Адрес   Команда
...
2000    BR 2500
2001    ---
2002    ---
...
2500    ; сюда будет передано управление по команде BR 2500
...
```

(точка с запятой «;» – признак начала строки с комментарием; многоточия «...» указывают на пропущенные фрагменты программы; строки «---» означают команды или данные, которые в данном случае не представляют для нас интереса).

Такой способ представления информации, когда слева представлены адреса памяти, а справа – команды, называется *картой памяти*; мы будем широко им пользоваться, так что рекомендуем отнестись к нему внимательно.

Если бы по адресу 2000 стояла не команда BR 2500, а какая-то другая команда, то она была бы исполнена, и новое значение PC стало бы равным 2001, т. е. адресу следующей по порядку команды. Но по адресу 2000 содержится именно команда безусловного перехода, которая предписывает изменить содержимое PC, после чего [PC]=2500. Устройство управления процессора считывает число 2500 и передает управление по этому адресу.

Теперь несколько слов о способах адресации. *Способ адресации* – это указание на то, как определить адрес ячейки памяти. В нашем первом примере мы использовали *непосредственную адресацию*, при которой адрес ячейки памяти был указан в виде числа непосредственно в команде. Но это не единственный способ. Как правило, в ассемблерах реализуется множество способов адресации (например, индексная или адресация с использованием базы), но нам понадобится еще только один способ адресации – косвенный. *Косвенная адресация* позволяет обращаться к ячейке памяти путем указания адреса этой ячейки в другой ячейке. Это звучит несколько туманно, но простой пример продемонстрирует, что это такое. Пусть ячейка памяти с адресом 2500 хранит (содержит) число 6000, т. е. [2500]=6000. Если мы напишем BR 2500, то, как мы уже видели, управление будет передано по адресу 2500 (это, как мы уже знаем, непосредственная адресация). Но если мы напишем чуть иначе: BR @2500, то устройство управления сначала обратится к ячейке памяти с адресом 2500, прочтет содержимое этой ячейки (т. е. число 6000) и, наконец, передаст управление по адресу 6000. Это напоминает передачу конверта в другом конверте. Получатель внешнего конверта вскрывает его, находит внутри другой конверт с адресом и передает этот конверт по указанному на нем адресу:

```
Адрес   Команда
...
2000    BR @2500
2001    ---
2002    ---
...
2500    6000
...
6000    ; сюда будет передано управление по команде BR @2500
...
```

Поначалу косвенная адресация может показаться сложной и несколько надуманной конструкцией, но очень скоро мы убедимся, что это не так; косвенная адресация существенно увеличивает гибкость и «освобождает» программы от необходимости явного указания адресов. Например, чтобы в последнем примере изменить адрес передачи управления с адреса 6000 на адрес 12800, достаточно изменить [2500] (содержимое ячейки памяти с адресом 2500) с 6000 на 12 800. При этом сама команда BR @2500 останется без изменений. Вскоре мы увидим многочисленные примеры использования кос-

венной адресации и сможем оценить предоставляемые косвенной адресацией возможности.

Наконец, подготовительные объяснения можно считать завершенными, и мы можем перейти к подпрограммам.

Вне зависимости от используемого языка программирования, в основе механизма управления подпрограммами лежит одна и та же схема из двух шагов:

- шаг 1 – передача управления на начало подпрограммы;
- шаг 2 – когда подпрограмма завершит свою работу, возобновляется работа той части программы, откуда была вызвана подпрограмма.

Другими словами, подпрограммы не являются самостоятельными программными единицами; они кем-то вызываются (эти «кто-то» могут быть основной программой, другими подпрограммами и даже той же самой подпрограммой). Затем подпрограмма начинает выполняться. Наконец, когда работа подпрограммы завершается, она должна вернуть управление туда, откуда она была вызвана. Если во время чтения книги мы слышим телефонный звонок, мы запоминаем с помощью закладки место, где нас прервали, и отвечаем на звонок (звонок – это вызов подпрограммы, а ответ – исполнение подпрограммы). По окончании разговора мы возвращаемся (возврат управления) к книге и продолжаем читать с того места, на котором нас прервали.

Фактически события такого рода называются прерываниями, а подпрограммы, которые должны быть выполнены в таких случаях, – обработчиками прерываний. В обычных языках программирования программист надежно «огражден» от необходимости обработки прерываний; прерывания – это «забота» операционной системы. Прерывания – это сложная и деликатная тема, поскольку именно через прерывания реализованы такие функции, как вывод на экран, работа с дисками и сетью, работа клавиатуры и мыши и т. п. Как правило, обработчики прерываний разрабатываются на ассемблерах высококвалифицированными программистами. Но концептуально обработчики прерываний – это те же самые подпрограммы.

Самый простой способ передать управление подпрограмме – выполнить безусловный переход по адресу ее начала. Наши первые примеры именно это фактически и делали (первый – с непосредственной, второй – с косвенной адресацией). Таким образом, первый шаг схемы механизма подпрограмм оказалось выполнить несложно. А как быть со второй частью схемы – возвратом из подпрограммы?

Пока, к сожалению, никак: вернуться на прежнее место и продолжить выполнение с команды по адресу 2001 невозможно, т. к. подпрограмма не «знает», откуда она была вызвана, и, следовательно, адрес 2001 ей неизвестен. И вот это – настоящая проблема.

Итак, ближайшая задача понятна – обеспечить возврат из подпрограммы, что эквивалентно запоминанию адреса, на который нужно передать управление по окончании работы подпрограммы. Этот адрес настолько важен, что имеет собственное имя – *адрес возврата* (return address).

Как подпрограмма может «узнать» этот адрес? Первое решение, которое приходит на ум, – запомнить адрес возврата либо в отдельном регистре процессора (обычно он называется *регистром связи*), либо в какой-то ячейке памяти. Принципиальных отличий в этих способах нет, но, как правило, выделять отдельный регистр для хранения адреса возврата слишком расточительно, т. к. количество регистров процессора ограничено и «расходовать» их нужно экономно. Поскольку количество ячеек памяти на много порядков превышает количество регистров, то мы остановимся на ячейке памяти. Для этого нам нужно выделить определенную ячейку памяти для хранения в ней адреса возврата. Это может быть почти любая ячейка памяти; главное требование к ней простое – следует воздержаться использовать эту ячейку для любых других целей, за исключением хранения адреса возврата из подпрограмм. Пусть для определенности это будет ячейка с адресом 1000.

Наш пример теперь можно переписать так:

```
Адрес   Команда
...
1000    ; место для хранения адреса возврата
...
        ; вызов подпрограммы
2000    BR 2500
2001    ---
2002    ---
...
2500    ; начало подпрограммы
...
        ; возврат из подпрограммы
2615    BR @1000
...
```

Разберемся, что тут происходит. Прежде всего мы договорились выделить ячейку памяти с адресом 1000 для хранения адреса возвра-

та. По адресу 2000 находится безусловный переход к началу подпрограммы (по адресу 2500). Прежде чем этот переход будет осуществлен (т. е. до того, как [PC] станет равным 2500) и подпрограмма начнет выполняться, нужно каким-то образом запомнить адрес возврата, т. е. адрес 2001, в ячейке с адресом 1000. Допустим, что мы это умеем делать (как именно – сейчас не важно, но мы запоминаем число 2001 в ячейке с адресом 1000, т. е. [1000]=2001). Теперь переходим к подпрограмме (она располагается в диапазоне адресов с 2500 по 2615). Адрес начала подпрограммы (2500) называется *точкой входа* в подпрограмму; адрес окончания подпрограммы (2615) – *точкой возврата*, или *точкой выхода* из подпрограммы.

А теперь – маленький фокус! В точке выхода из подпрограммы мы видим команду BR @1000. Это безусловный переход по косвенному адресу (вот нам и пригодилась косвенная адресация), т. е. по адресу, который хранится в ячейке памяти с адресом 1000. А что там хранится? Ну, конечно, 2001. Так мы возвращаемся к основной программе.

Чуть выше мы говорили о том, что перед передачей управления подпрограмме мы каким-то образом умеем сохранять адрес возврата в специально выделенной ячейке памяти (в нашем примере в ячейке с адресом 1000). Этот момент требует внимания, т. к. от этого зависит, сможет ли подпрограмма вернуть управление по окончании своей работы.

Одно из решений состоит в следующем: адрес возврата можно запомнить непосредственно перед вызовом подпрограммы. Мы знаем, что команда вызова подпрограммы располагается по адресу 2000, следовательно, нам известен и адрес возврата из подпрограммы (т. е. адрес 2000+1=2001). Поэтому мы можем сохранить адрес возврата в ячейке памяти с адресом 1000, а затем передать управление подпрограмме. Это означает, что перед вызовом подпрограммы необходимо, чтобы [1000]=2001. Для этого в системах команд практически всех процессоров имеется команда пересылки MOV (от move), с помощью которой можно обновить содержимое заданной ячейки памяти. Это вполне работоспособное решение, но по причинам, которые станут понятными чуть ниже, мы им не воспользуемся.

Другое решение состоит в следующем. Введем в нашу систему команд новую команду и назовем ее JSR (от Jump to SubRoutine). Это фактически тот же самый безусловный переход BR, но только с одним отличием: перед тем как осуществлять передачу управления подпрограмме, команда JSR запоминает адрес возврата по заранее определенному адресу (в нашем примере по адресу 1000). Итак, наш пример будет теперь выглядеть так:

```
Адрес    Команда
...
1000     ; место для хранения адреса возврата
...
         ; запоминание адреса возврата и вызов подпрограммы
2000     JSR 2500
2001     ---
2002     ---
...
2500     ; начало подпрограммы
...
         ; возврат из подпрограммы
2615     BR @1000
...
```

Команда JSR 2500 сначала запоминает адрес возврата в ячейке с адресом 1000, а потом передает управление по указанному в ней адресу. Сам адрес возврата (т. е. 2001) известен – это, как и раньше, адрес, непосредственно следующий за адресом, по которому располагается команда JSR.

Что же, задача решена? В какой-то мере, да – решена. Но это решение все же плохое. Недостаток этого решения кроется в способе хранения адреса возврата. Рассмотрим небольшой пример, который фактически приводит к необходимости отбросить этот метод решения как совершенно непригодный:

```
Адрес    Команда
...
1000     ; место для хранения адреса возврата
...
         ; запоминание адреса возврата и вызов первой подпрограммы
2000     JSR 2500
2001     ---
2002     ---
...
2500     ; начало первой подпрограммы
2501     ---
...
         ; запоминание адреса возврата и вызов второй подпрограммы
2600     JSR 4500
2601     ---
...
         ; возврат из первой подпрограммы
2615     BR @1000
```

```
...
4500    ; начало второй подпрограммы
4501    ---
...
        ; возврат из второй подпрограммы
4580    BR @1000
...
```

Посмотрим, что тут происходит. Начало нам уже знакомо – запоминаем адрес возврата ([1000]=2001) и переходим к подпрограмме по адресу 2500. Но внутри этой подпрограммы (точнее, по адресу 2600) происходит вызов другой подпрограммы. Исполнение текущей (первой) подпрограммы временно приостанавливается. Каковы наши дальнейшие действия? Запоминаем адрес возврата ([1000]=2601) и переходим к подпрограмме по адресу 4500. Когда эта (вторая) подпрограмма доходит до точки возврата по адресу 4580, выполняется возврат в первую подпрограмму, и ее исполнение возобновляется с адреса 2601. Наконец, первая подпрограмма доходит до точки возврата по адресу 2615, она должна вернуть управление. Вот только куда? Логически рассуждая, по адресу 2001, а практически? Содержимое ячейки памяти с адресом 1000 было изменено в тот момент, когда мы вызывали вторую подпрограмму, и нужный нам теперь адрес возврата (2001) попросту исчез. Его нет, и поэтому команда BR @1000 вернет управление по адресу ... 2601, т. е. внутрь самой себя. А это совсем не то, что ожидалось: первая подпрограмма попросту войдет в бесконечный цикл, исполняя команды по адресам с 2601 по 2615. Мы никогда не вернемся к выполнению основной программы по адресу 2001, т. к. этот адрес был фактически уничтожен вызовом второй подпрограммы.

Мы обращаем особое внимание на этот пример и предлагаем внимательно с ним разобраться: он весьма поучителен и демонстрирует, как легко и просто нарушить работу программы. Бесконечные циклы весьма коварны, т. к. далеко не всегда бывает очевидно, что именно привело к зацикливанию программы.

Итак, решение с выделенной ячейкой памяти для хранения адреса возврата работает, но налагает на вызовы подпрограмм строгое (можно сказать, фундаментальное) ограничение: нельзя вызывать другие подпрограммы до тех пор, пока не заверен возврат из текущей (активной) подпрограммы. Каждый вызов подпрограммы затирает предыдущий адрес возврата. Это слишком жесткое ограничение, и мы не можем его принять. Надо искать что-то другое...

А что, если мы будем хранить адрес возврата не в выделенной ячейке памяти, где он будет затираться новыми значениями всякий раз, когда происходит вызов подпрограммы, а в самой подпрограмме? Это, на первый взгляд, абсурдное решение, но не будем торопиться.

Прежде всего нам может оказаться полезной команда NOP (no operation), основное назначение которой – ничего не делать. Эта команда как бы «говорит» процессору – «здесь можно отдохнуть». Посмотрите на следующий пример:

```
Адрес   Команда
...
        ; вызов подпрограммы
2000    JSR 2500
2001    ---
2002    ---
...
        ; здесь будет запоминаться адрес возврата
2500    NOP
...
        ; возврат из подпрограммы
2615    BR @2500
...
```

Это выглядит немножко странно, но давайте сначала разберемся. Команда JSR 2500 по адресу 2000 передает управление по адресу начала подпрограммы. Подпрограмма начинается с команды NOP. Вместо того чтобы запоминать адрес возврата в выделенной ячейке памяти (например, в ячейке памяти с адресом 1000, как это было раньше), команда JSR записывает адрес возврата вместо команды NOP. Непосредственно после вызова подпрограммы (но до того, как будет выполнен возврат из нее) карта памяти будет выглядеть так:

```
Адрес   Команда
...
        ; вызов подпрограммы
2000    JSR 2500
2001    ---
2002    ---
...
        ; здесь находится адрес возврата
2500    2001
...
        ; возврат из подпрограммы
2615    BR @2500
...
```

Подпрограмма начинает исполняться с адреса, следующего за адресом команды NOP, т. е. с адреса 2501. Когда управление получит команда по адресу 2615, то адрес возврата будет извлечен из ячейки памяти с адресом 2500 (т. е. будет извлечено значение 2001), и команда BR @2500 вернет управление основной программе. Назначение команды NOP состоит в том, чтобы зарезервировать место для адреса возврата; в принципе, можно записывать в эту ячейку совершенно произвольное значение (ведь оно все равно будет перезаписано адресом возврата), но вариант с NOP выглядит аккуратнее. Смысл NOP только в том, чтобы зарезервировать место для хранения адреса возврата и ничего более. Выглядит хитро, но будет ли это работать в случае вызова одной подпрограммы из другой? Проверим:

```
Адрес    Команда
...
         ; вызов первой подпрограммы
2000     JSR 2500
2001     ---
2002     ---
...
         ; здесь запоминается адрес возврата первой подпрограммы
2500     NOP -> 2001
2501     ---
...
         ; вызов второй подпрограммы
2600     JSR 4500
2601     ---
...
         ; возврат из первой подпрограммы
2615     BR @2500
...
         ; здесь запоминается адрес возврата второй подпрограммы
4500     NOP -> 2601
4501     ---
...
         ; возврат из второй подпрограммы
4580     BR @4500
...
```

Здесь мы, для простоты отслеживания, слегка изменили форму представления команд по адресам 2500 и 4500, чтобы акцентировать внимание на том, что до вызова подпрограмм в этих ячейках находилась команда NOP, а после вызова – адреса возвратов. Проследим, что тут происходит. Сначала вызывается первая подпрограмма (та,

что расположена по адресам 2500–2615). Адрес возврата (2001) запоминается в первой ячейке памяти этой подпрограммы, т. е. по адресу 2500. Первая подпрограмма еще не завершила своей работы и вызывает вторую подпрограмму (ту, что занимает адреса 4500–4580). Адрес возврата (2601) запоминается в первой ячейке памяти вызванной подпрограммы, т. е. по адресу 4500. Итак, имеем [2500]=2001 и [4500]=2601.

Когда вторая подпрограмма завершает свою работу (адрес 4580), она возвращает управление по адресу 2601. Первая подпрограмма возобновляет свою работу. Когда первая подпрограмма завершает свою работу (адрес 2615), она возвращает управление по адресу 2001. Обе подпрограммы отработали, корректно вернули управление, и теперь программа продолжает выполнение с адреса 2001. Как говорят математики, «что и требовалось доказать».

Итак, давайте подведем некоторые итоги. Наша задача была такой: обеспечить механизм возврата из подпрограммы в программу (или в другую подпрограмму), из которой она была вызвана. Метод с использованием универсальной выделенной ячейки памяти для хранения адреса возврата крайне ограничивает наши возможности и диктует совершенно неприемлемую дисциплину: до вызова новой подпрограммы необходимо полностью завершить работу текущей. Метод, при котором адрес возврата хранится в самой подпрограмме, гораздо лучше. Он позволяет делать вызовы подпрограмм из других подпрограмм. Правда, для хранения адреса возврата в подпрограмме приходится резервировать дополнительную ячейку памяти, но это представляется незначительной платой за столь широкие возможности.

Перед тем как продолжить рассказ, давайте вернемся к последнему примеру и, отбросив в сторону все «лишнее», внимательно приглядимся к тому, что мы делали. Последовательность вызовов подпрограмм и возвратов из них выглядела так:

```
JSR   2500
JSR   4500
BR   @4500
BR   @2500
```

т. е. вызов первой подпрограммы, вызов второй подпрограммы, возврат из второй подпрограммы, возврат из первой подпрограммы. Ничего не напоминает? Как будто бы пары команд JSR/BR ведут себя как вложенные скобки.

Неужели это...

Подпрограммы: появляется стек

Да-да, мы не ошиблись – это что-то, похожее на стек. И опять уже знакомая нам последовательность: то, что началось первым, заканчивается последним. А раз вызовы и возвраты из подпрограмм соответствуют дисциплине LIFO, то этой же дисциплине должны подчиняться и адреса возвратов. Следовательно, для их запоминания можно воспользоваться стеком. Назовем такой стек *стеком возвратов*.

Очевидно, что стек возвратов должен существовать до того, как программа будет запущена на исполнение. Один из вопросов, который нужно решить на этом этапе, – глубина стека, т. е. количество элементов, которые стек может вместить. Если мы говорим об обычных (не рекурсивных) подпрограммах, то обычно оказывается достаточно стека на несколько десятков элементов (адресов). В случае рекурсивных подпрограмм (речь о которых пойдет в следующем разделе), особенно предназначенных для обработки сложных структур данных (например, деревьев), глубина стека может составлять сотню, тысячу и более элементов. Резервировать стек такого объема – как правило, непозволительная роскошь: вполне может случиться, что фактически будет использована лишь незначительная часть отведенного под стек адресного пространства. Обычно в этом случае используется такой способ реализации стека, при котором память для него выделяется динамически, т. е. по мере надобности. В этом разделе мы не будем касаться реализации стеков (подробнее об этом см. в одном из приложений), но этот вопрос обязательно должен быть решен. Пока же мы будем просто считать, что такой стек есть.

Следующий вопрос относится уже к механизму реализации подпрограмм: как обеспечить занесение в стек адреса возвратов при вызове подпрограмм и, обратно, как обеспечить выборку из стека сохраненных в нем адресов возвратов по окончании работы подпрограмм?

Первая проблема решается путем небольшого изменения семантики, т. е. поведения, команды JSR (переход к подпрограмме). В предыдущем разделе команда JSR сохраняла адрес возврата в выделенной ячейке тела подпрограммы (а именно – в самой первой). Теперь же эта команда будет проталкивать адрес возврата в стек возвратов. Вторая проблема чуточку сложнее. Вспомните, что в предыдущем случае возврат из подпрограммы обеспечивался командой BR @address, где address и есть ранее сохраненный адрес возврата. Теперь же адрес возврата нужно выбрать из стека (если быть точным – из ячейки, на которую указывает его вершина). Но это не все. После возврата из

подпрограммы состояние стека должно быть скорректировано; мы уже использовали текущий адрес возврата (на вершине стека), и он нам больше не нужен. Значит, нужно обеспечить, чтобы на вершине стека теперь был адрес возврата подпрограммы, вызванной раньше текущей. Иначе говоря, стек должен быть уменьшен на один элемент, и вершина стека должна указывать на новый адрес возврата. Команда BR сама по себе этого делать не может. Можно было бы, конечно, как и в случае команды JSR, изменить ее семантику так, чтобы при возврате из подпрограммы одновременно корректировался и стек, но это плохое решение, и вот почему. Команда BR, как правило, используется не для возврата из подпрограмм, а для организации обычных безусловных переходов (аналог goto, о чем мы говорили ранее). Это ее основное предназначение. Если команда BR при любом своем использовании будет обновлять состояние стека, то ни к чему хорошему это не приведет. Иначе говоря, команда BR делает слишком много. Чтобы поправить ситуацию и уберечься от неконтролируемых изменений стека, в систему команд обычно вводится отдельная команда, предназначенная исключительно для обеспечения возвратов из подпрограмм. Введем такую команду и мы. Назовем эту команду RET (return). Действие этой команды заключается в следующем: прочесть адрес возврата на вершине стека возвратов, вытолкнуть этот адрес возврата из стека (после чего вершина стека будет указывать на адрес возврата подпрограммы, вызванной раньше) и передать управление по уже известному адресу возврата. Рассмотрим пример:

```
Адрес   Команда                                 Стек
...
        ; вызов подпрограммы
2000    JSR 2500                                | 2001
2001    ---
2002    ---
...
2500    ; начало подпрограммы                   | 2001
...
        ; возврат из подпрограммы
2615    RET
...
```

Допустим, что перед вызовом подпрограммы стек был пуст (это совершенно не обязательное условие, но так проще отслеживать его состояние). При вызове по адресу 2000 подпрограммы происходит следующее: 1) адрес возврата (2001) проталкивается в стек, и 2) осу-

ществляется передача управления на адрес начала подпрограммы (2500). Подпрограмма начинает выполняться, и, наконец, управление получает команда RET. При этом 1) адрес возврата, находящийся на вершине стека возвратов (2001), запоминается, 2) этот адрес возврата выталкивается из стека, и 3) осуществляется передача управления на адрес возврата (т. е. на адрес 2001). Сама передача управления, напоминаем, сводится к изменению содержимого PC. Стек возвратов возвращается в то же самое состояние, в котором он был до вызова подпрограммы.

Так, пока все идет неплохо. А что с вложенными вызовами подпрограмм?

```
Адрес   Команда                                        Стек
...                                                    ├
2000    JSR 2500                                       ├ 2001
2001    ---                                            ├
2002    ---
...
        ; начало первой подпрограммы
2500    ---                                            ├ 2001
...
2600    JSR 4500                                       ├ 2001 2601
2601    ---
...
2615    RET
...
        ; начало второй подпрограммы
4500    ---                                            ├ 2001 2601
...
4580    RET
...
```

Проанализируем, что тут происходит, по порядку:

- вызов (по адресу 2000) первой подпрограммы; адрес возврата (2001) проталкивается в стек;
- первая подпрограмма начинает исполняться;
- вызов (по адресу 2600) второй подпрограммы; адрес возврата (2601) проталкивается в стек;
- вторая подпрограмма начинает исполняться;
- управление получает команда RET (по адресу 4580);
- адрес возврата с вершины стека (2601) запоминается и затем выталкивается из стека;

- передача управления на адрес 2601; работа второй подпрограммы полностью завершена;
- возобновление (с адреса 2601) работы первой подпрограммы;
- управление получает команда RET (по адресу 2615);
- адрес возврата с вершины стека (2001) запоминается и затем выталкивается из стека;
- передача управления на адрес 2001; работа первой подпрограммы полностью завершена;
- стек возвращается в исходное состояние (до начала вызова первой подпрограммы);
- возобновляется работа основной программы.

Итак, стек весьма логичным и изящным способом позволяет организовать вызовы и возвраты из подпрограмм.

Однако описание работы стека при этом выглядит многословно и неуклюже. Конечно, стековая диаграмма в правой части примера определенно полезна, но надо помнить, что это – всего лишь часть описания. Давайте введем некоторые обозначения, которые позволят нам рассуждать о подпрограммах (и, как будет видно в дальнейшем, не только о подпрограммах) и стеках более компактно. Для этого введем специальный регистр, который будет обозначаться как SP (stack pointer) – *указатель стека*. SP всегда указывает на вершину стека (в пустом стеке вершина стека, т. е. SP, совпадает с его дном).

При проталкивании адреса возврата в стек значение SP корректируется: SP=SP-1, и новое значение [SP]=адресу возврата. Почему SP уменьшается на 1? Вспомним, ранее мы условились, что стек растет от больших адресов к меньшим, т. е. в сторону младших адресов памяти (к началу памяти). Поэтому при проталкивании в стек нового элемента значение SP должно быть уменьшено.

При выталкивании адреса возврата из стека значение SP вновь корректируется, но уже в «обратную» сторону, т. е. в сторону старших адресов памяти, и SP=SP+1, а [SP]=предыдущему адресу возврата. Давайте перепишем работу только что рассмотренного примера в новых обозначениях:

- вызов (по адресу 2000) первой подпрограммы; SP=SP-1, [SP]=2001;
- первая подпрограмма начинает исполняться;
- вызов (по адресу 2600) второй подпрограммы; SP=SP-1, [SP]=2601;
- вторая подпрограмма начинает исполняться;
- управление получает команда RET (по адресу 4580);

- адрес возврата=[SP] (т. е. 2601); SP=SP+1;
- передача управления на адрес 2601; работа второй подпрограммы полностью завершена;
- возобновление (с адреса 2601) работы первой подпрограммы;
- управление получает команда RET (по адресу 2615);
- адрес возврата=[SP] (т. е. 2001); SP=SP+1;
- передача управления на адрес 2001; работа первой подпрограммы полностью завершена;
- стек возвращается в исходное состояние.

Во многих компьютерных архитектурах регистр, подобный указателю стека SP, действительно предусмотрен. Можно, конечно, реализовать стек и программно, если такого регистра нет.

Сравнительно с предыдущими вариантами вызова и возврата из подпрограмм (с выделенной ячейкой для всех подпрограмм и специальной ячейкой в теле самой подпрограммы) последний вариант со стеком выглядит несколько тяжеловесно. Потребовалось ввести новую команду (RET) и новый регистр (SP). И новая команда, и указатель стека должны взаимодействовать между собой совершенно определенным образом, что несколько снижает наглядность. Но это впечатление обманчиво. Все операции по «манипулированию» состоянием стека (что сводится фактически к работе с регистром SP) весьма просты и, что называется, «спрятаны под капот».

Для простейших случаев рассмотренные ранее варианты (особенно второй – с сохранением адреса возврата в теле подпрограммы), возможно, и хороши. Но дело в том, что простейших случаев практически не бывает. Здесь уместна некоторая аналогия с задачей анализа скобочных структур. Метод со счетчиком прост, нагляден, но на этом все его преимущества и заканчиваются: в общем случае скобочные структуры методом со счетчиком разобрать практически невозможно. Введение стека решило разом все проблемы, и теперь, какие бы типы скобок – даже самые «экзотические» – нам не встретились, все они будут разобраны единообразно и совершенно автоматически. Так же точно дело обстоит и в задаче с подпрограммами.

Напоследок давайте обновим терминологию. Вызов подпрограммы командой JSR с одновременным проталкиванием в стек возвратов адреса возврата и уменьшением значения указателя стека SP принято называть *прологом*. Возврат из подпрограммы командой RET с одновременным выталкиванием из стека возвратов адреса возврата и увеличением значения указателя стека SP принято называть *эпилогом*.

В общем случае пролог и эпилог могут сопровождаться и другими действиями, но описанные – главные и обязательные.

Разделы, посвященные подпрограммам, нельзя назвать простыми. Это и понятно: здесь мы имеем дело не со статичными скобочными структурами, которые записываются на этапе составления программ лишь однажды и считываются из исходных кодов, а с динамическими вызовами подпрограмм. В общем случае порой невозможно точно сказать, когда, какие и как подпрограммы будут вызваны; таких вызовов может и не быть ни одного, а может быть и много тысяч. В следующем разделе, используя уже накопленные нами знания, мы обратимся к практически важному случаю рекурсивных вызовов подпрограмм.

Подпрограммы: рекурсия

А теперь давайте посмотрим, что произойдет, если подпрограмма вызывает саму себя, т. е. рекурсивные вызовы. Заранее стоит предупредить: этот раздел будет достаточно сложным.

Здесь мы продолжим обсуждение темы управления подпрограммами (вызовы и возвраты из них). Метод, при котором адрес возврата запоминался в специально выделенной для этой цели ячейке памяти, мы не рассматриваем по очевидным причинам.

Сначала рассмотрим случай с сохранением адреса возврата в теле самой подпрограммы (т. е. без использования стека):

```
Адрес   Команда
...
        ; вызов подпрограммы
2000    JSR 2500
2001    ---
2002    ---
...
        ; здесь запоминается адрес возврата подпрограммы
2500    NOP
2501    ---
...
        ; рекурсивный вызов подпрограммы
2600    JSR 2500
2601    ---
...
        ; возврат из подпрограммы
2615    BR @2500
...
```

Посмотрим, что тут происходит. Поначалу все идет, как обычно. По адресу 2000 происходит вызов подпрограммы, начинающейся с адреса 2500. Адрес возврата (т. е. 2001) записывается в ячейке памяти с адресом 2500. Запомним это хорошенько. Потом по адресу 2600 (который является частью самой подпрограммы) происходит рекурсивный вызов этой же подпрограммы, и подпрограмма рекурсивно начинает выполняться с самого начала, т. е. с адреса памяти 2500.

Куда должен быть осуществлен возврат по окончании рекурсивного вызова? В соответствии с уже установленным порядком это адрес 2601. Где должен быть сохранен этот адрес возврата (т. е. где нужно сохранить адрес 2601)? Он должен быть сохранен по адресу 2500, т. к. мы договорились, что первая ячейка памяти тела подпрограммы служит местом сохранения адреса возврата. Но вспомните, когда мы вызывали подпрограмму в первый раз (по адресу 2000), мы записали по адресу 2500 число 2001. После рекурсивного вызова это число будет перезаписано новым адресом возврата (а именно адресом 2601). Итак, содержимое ячейки памяти по адресу 2500 принимает последовательно два значения: сначала 2001, затем 2601. Что из этого следует? Ничего хорошего – адрес возврата в основную программу (2001) будет потерян, и мы не сможем вернуться из подпрограммы в основную программу.

Это довольно тонкий момент, и мы предлагаем на некоторое время остановиться и еще раз перечитать предыдущий абзац.

Вот как будет выполняться эта программа (мы приводим список только тех адресов, которые связаны с вызовами подпрограмм и возвратами из них):

```
2000    ; основная программа; вызов подпрограммы
2500    ; сохранить адрес возврата (2001)
2501    ; начало исполнения подпрограммы
2600    ; первый рекурсивный вызов подпрограммы
2500    ; сохранить адрес возврата (2601)
2501    ; начало исполнения подпрограммы (второй рекурсивный вызов)
2600    ; рекурсивный вызов подпрограммы
2500    ; сохранить адрес возврата (2601)
2501    ; начало исполнения подпрограммы (третий рекурсивный вызов)
...
```

Итак, мы видим последовательность адресов 2500-2501-2600, которая будет исполняться бесконечно. Это – еще одна форма бесконечного цикла, с которым мы уже встречались раньше. Очевидно, что в случае рекурсивных вызовов метод с сохранением адреса возврата

в первой ячейке памяти подпрограммы непригоден: мы никогда, ни при каких условиях не вернемся в основную программу, т. к. этот адрес утерян навсегда и безвозвратно. Подпрограмма будет исполняться и исполняться, и ее (а вместе с ней и основную программу) придется принудительно прервать. Итак, этот метод управления рекурсивными подпрограммами мы вынуждены признать непригодным. Но, может быть, стек (уже в который раз) нам поможет? Давайте посмотрим:

```
Адрес   Команда
...                                                ├
        ; вызов подпрограммы
2000    JSR 2500                                   ├ 2001
2001    ---
2002    ---
...
        ; начало подпрограммы
2500    ---                                        ├ 2001
2501    ---
...
        ; рекурсивный вызов подпрограммы
2600    JSR 2500                                   ├ 2001 2601
2601    ---
...
        ; возврат из подпрограммы
2615    RET
...
```

Внимательно проследим за изменением стека возвратов. Прежде всего из основной программы (по адресу 2000) вызывается подпрограмма. Адрес возврата (2001) запоминается в стеке возвратов. Подпрограмма (она, как и прежде, начинается с адреса 2500) начинает выполняться. По адресу 2600 происходит рекурсивный вызов подпрограммы, и в стек возвратов добавляется новый адрес возврата (2601). Подпрограмма вновь начинает выполняться с адреса 2500. Но обратите внимание – адрес возврата в основную программу (т. е. 2001) никуда не пропал – он лежит в стеке возвратов! Не исключено, что будут произведены еще и рекурсивные вызовы, и стек возвратов будет накапливать адреса в следующей последовательности:

```
├ → ├ 2001 → 2001 2601 → ├ 2001 2601 2601 → ├ 2001 2601 2601 2601
```

(здесь выполнены три рекурсивных вызова подпрограммы). Глубина стека возвратов, очевидно, будет увеличиваться на один элемент при каждом рекурсивном вызове подпрограммы.

Что произойдет при достижении команды возврата RET по адресу 2615? Пусть стек возвратов находится в только что указанном состоянии. А произойдет следующее (вспоминаем о регистре указателя стека SP, который был введен нами в предыдущем разделе, посвященном подпрограммам):

- адрес возврата = [SP] (т. е. 2601); SP=SP+1;
- передача управления на адрес 2601 (возврат из третьего рекурсивного вызова);
- управление вновь получает команда RET;
- адрес возврата = [SP] (т. е. 2601); SP=SP+1;
- передача управления на адрес 2601 (возврат из второго рекурсивного вызова);
- управление вновь получает команда RET;
- адрес возврата = [SP] (т. е. 2601); SP=SP+1;
- передача управления на адрес 2601 (возврат из первого рекурсивного вызова);
- управление вновь получает команда RET;
- адрес возврата = [SP] (т. е. 2001); SP=SP+1;
- возврат в основную программу; рекурсивный вызов подпрограммы полностью завершен.

Итак, стек возвратов обеспечил сохранение всех адресов возврата и последовательный возврат из рекурсивных подпрограмм в основную программу.

Может показаться, что тут что-то не так. Если вдуматься, то здесь действительно многое требует пояснений. Сейчас мы этим и займемся.

Прежде всего мы продемонстрировали три рекурсивных вызова, и максимальная глубина стека возвратов (с учетом возврата в основную программу) составила весьма небольшую величину, равную 4. А что, если рекурсивных вызовов будет не три, а тридцать, триста, три тысячи и больше? Это вполне возможно; более того – это отнюдь не редкость. Если стек возвратов не достаточно глубок, то очень быстро рекурсивные вызовы полностью его заполнят. Что произойдет в этом случае?

Если среда исполнения программы, например операционная система или виртуальная машина, контролирует переполнение (overflow) стека возвратов (а так обычно это и происходит), то произойдет аварийная остановка программы с выводом соответствующего сообщения. Если же такого контроля нет, то стек возвратов начнет «захватывать» память, принадлежащую другим программам. Ничего хорошего в этом,

конечно, нет. Более того, если рекурсивных вызовов будет слишком много, то рано или поздно исчерпается вся доступная память, и работа всего компьютера завершится полным крахом. Вопросы регулирования глубины стека возвратов (впрочем, как и всех других стеков) требуют отдельного обсуждения и относятся, скорее, к реализации стека. Кое-что об этом можно прочесть в одном из приложений.

Но если глубина стека возвратов – вопрос, в общем, технический (хотя и очень важный), то следующая проблема находится целиком и полностью в ведении программиста. Эту проблему можно сформулировать так: когда следует прекратить рекурсивные вызовы и как инициировать процесс возврата из них? В последнем примере мы молчаливо предполагали, что рекурсивные вызовы когда-то будут прекращены, но это, конечно, не решение проблемы. Множество попыток разработки рекурсивных программ потерпело полное фиаско именно потому, что программисты не обеспечили корректного завершения рекурсии. Именно поэтому рекурсивные программы с таким трудом находят признание среди программистов (и отнюдь не всегда только начинающих). Этой проблеме будет посвящена оставшаяся часть данного раздела.

Давайте посмотрим два примера рекурсии (язык программирования Java):

```
int factorial (int n) {
    if (n <= 0) return 1;
    return (n * factorial (n - 1));
}

int sum (int a, int b) {
    int s = a;
    if (b == 0) return s;
    return (sum (a, b - 1) + 1);
}
```

(здесь мы специально подчеркнули рекурсивные вызовы).

Первый пример – классический пример рекурсивного вычисления факториала (n!=1*2*...*n). Второй пример – сумма двух целых чисел a и b. Сумма образуется прибавлением к первому слагаемому (числу a) стольких единиц, сколько их содержится во втором слагаемом (в числе b). То есть для сложения, скажем, a=10 и b=3, к 10 добавляются три единицы, составляющие число b.

Конечно, на практике такой способ сложения чисел не используется, и мы приводим его лишь как наглядный пример рекурсии.

Кстати, второй из примеров позволяет наглядно убедиться в том, что проблема переполнения стека может возникнуть достаточно просто. Если мы передадим в качестве второго слагаемого (ему соответствует переменная b) достаточно большое число (например, 10 000) и попробуем выполнить метод, то очень скоро будет сгенерировано сообщение об ошибке, и выполнение программы прекратится. Эта же проблема возникнет и в том случае, если передать отрицательное значение. В обоих случаях произойдет тривиальное переполнение стека возвратов (после шести с небольшим тысяч рекурсивных вызовов на компьютере автора).

Напомним еще раз: отличительная особенность рекурсии в том, что подпрограмма может вызвать саму себя. Конечно, наши примеры на языке программирования Java можно (а практически даже нужно) переписать без использования рекурсии, но во многих случаях рекурсия оказывается удобным и естественным способом реализации задач.

В наших примерах оценить точное количество рекурсивных вызовов легко, но порой такая оценка затруднена, и количество рекурсивных вызовов заранее неизвестно. Рекурсия практически незаменима, когда речь идет об обработке сложных структур данных (например, списков, деревьев или графов). При использовании рекурсии важно обеспечить ее завершение, т. е. в программе должно быть условие, при котором рекурсивные вызовы прекращаются. Такое условие мы так и будем называть – *условием завершения* рекурсии. В наших примерах подобным условием была проверка на 0: параметр (n для факториала и b для суммирования) при каждом рекурсивном вызове уменьшается на 1. Рано или поздно n и b уменьшатся до 0, и рекурсия должна будет завершиться.

В правильно написанной программе всякая рекурсия должна рано или поздно завершиться, т. е. в такой программе должно содержаться условие завершения рекурсии. В наших примерах условие завершения рекурсии состояло в проверке на 0; в других случаях это условие может быть иным, но такое условие в любом случае должно быть, иначе неизбежно возникновение проблем, которые приведут к тому, что программа либо никогда не завершится, либо завершится с ошибкой (вкратце мы об этом уже упомянули выше).

Если условие завершения рекурсии выполняется, то рекурсивные вызовы прекращаются; в противном случае происходит новый рекурсивный вызов. Когда в программировании используется слово

«условие», то под этим подразумевается проверка чего-либо. Следовательно (мы вновь возвращаемся к нашему ассемблеру), нам необходима команда для проверки: продолжать рекурсивные вызовы или прекратить. Введем, в дополнение к уже имеющимся командам BR, JSR, RET и NOP, новую команду и назовем ее TSZ (от TeSt on Zero).

Команда TSZ работает так. Проверяется содержимое некоторой ячейки памяти. Если содержимое этой ячейки равно 0, то выполняется команда, непосредственно следующая за TSZ; если содержимое этой ячейки не равно 0, то команда, непосредственно следующая за TSZ, пропускается. Вот как может выглядеть фрагмент кода с использованием TSZ:

```
...
TSZ cond    ; если [cond] = 0
BR addr     ; то перейти по адресу addr
NOP         ; иначе продолжить
...
```

Сначала мы проверяем содержимое ячейки с адресом cond (от слова condition, т. е. «условие»). Если эта ячейка содержит 0, то выполняется команда BR addr и происходит передача управления по адресу addr. Если содержимое этой ячейки не равно 0, то выполняется следующая по порядку команда (т. е. в данном случае команда NOP).

Если эта конструкция содержится внутри подпрограммы, то, подставив вместо NOP вызов этой же самой подпрограммы (команду JSR), мы выполним ее рекурсивный вызов.

Команда TSZ напоминает простой вариант условного оператора IF-THEN-ELSE, хорошо знакомого нам по другим языкам программирования:

IF (cond == 0) THEN ... ELSE ...

с той только разницей, что команда TSZ «умеет» проверять лишь одно условие, а именно условие равенства нулю.

Перед тем как продолжить рассказ о рекурсии, введем еще одну команду. Это будет команда декремента (от decrement), т. е. уменьшения на 1 содержимого указанной ячейки памяти: DEC XXXX (здесь XXXX – адрес памяти). Вот небольшой пример использования команды DEC:

```
Адрес   Команда
...
1500    DEC 5000
...
5000    3
...
```

Содержимое ячейки памяти с адресом 5000 равно 3 (т. е. [5000]=3). Команда DEC 5000 по адресу 1500 уменьшает [5000] на 1, после чего [5000]=2.

Наконец, теперь мы готовы к тому, чтобы обсудить, как выполняются рекурсивные вызовы и, главное, как происходит возврат из них.

Вернемся к нашему языку ассемблера и рассмотрим следующую модификацию нашего примера:

```
Адрес   Команда
...                                                   ├
        ; вызов подпрограммы
2000    JSR 2500                                      ├ 2001
...
        ; начало подпрограммы
2500    TSZ 5000        ; проверка                    ├ 2001
2501    BR  2615        ; переход если 0
2502    DEC 5000        ; не 0
...
        ; рекурсивный вызов подпрограммы
2600    JSR 2500                                      ├ 2001 2601
2601    ---
...
        ; возврат из подпрограммы
2615    RET
...
        ; счетчик (условие завершения рекурсии)
5000    3
...
```

(обратите внимание на комментарии в подпрограмме).

Мы добавили несколько команд по адресам с 2500 по 2502. Сначала мы проверяем значение, хранящееся по адресу 5000. Напомним, что изначально [5000] = 3. Так как это значение не равно 0, то следующая команда по адресу 2501 (BR 2615) пропускается, и управление передается по адресу 2502, где происходит уменьшение значения по адресу 5000 на 1.

Следовательно, после выполнения этой команды [5000] = 2. Потом (по адресу 2600) идет рекурсивный вызов подпрограммы, и она вновь начинает, как и положено, выполняться с адреса 2500.

Затем повторяется та же самая последовательность: проверка содержимого ячейки с адресом 5000, его декремент (после чего [5000] = 1) и новый рекурсивный вызов. Наконец, после очередного декремента содержимого ячейки с адресом 5000 (теперь, очевидно, [5000] = 0) и проверки на 0 мы передаем управление по адресу 2615, т. е. на команду возврата из подпрограммы.

Эта команда осуществляет возврат из последнего рекурсивного вызова (передает управление по адресу 2601). После этого команда RET еще дважды осуществляет возврат из рекурсивных вызовов и, наконец, после того как все рекурсивные вызовы завершены, в стеке возвратов остается единственный адрес (2001) – адрес возврата в основную программу. Этот возврат выполняется, и программа продолжает выполняться с адреса 2001.

Вот и все – мы сделали три рекурсивных вызова и три возврата из них. Ячейка памяти с адресом 5000 выполняла в нашей программе роль счетчика, а условием завершения рекурсии было нулевое значение счетчика.

Таким образом, мы добились того, чего хотели, – наши подпрограммы теперь допускают рекурсию.

Последний пример можно переписать проще:

```
Адрес   Команда
...                                          ├
        ; вызов подпрограммы
2000    JSR 2500                             ├ 2001
...
        ; начало подпрограммы
2500    TSZ 5000        ; проверка           ├ 2001
2501    RET             ; если 0, то возврат из подпрограммы
2502    DEC 5000        ; не 0
...
        ; рекурсивный вызов подпрограммы
2600    JSR 2500                             ├ 2001 2601
2601    ---
...
        ; счетчик (условие завершения рекурсии)
5000    3
...
```

Не следует пользоваться этим приемом как стандартным правилом; это, скорее, рекомендация. Иногда этот прием может и не сработать – все зависит от решаемой задачи.

В рекурсивных вызовах самих по себе нет ничего загадочного и необычного. Это обычный и распространенный метод программирования. Единственная принципиальная тонкость, связанная с рекурсией, – обеспечить условие завершения, которое блокирует дальнейшие рекурсивные вызовы и начинает цепочку возвратов.

На этом мы на время заканчиваем обсуждение подпрограмм и рекурсивных вызовов, но позже к ним обязательно вернемся в связи с проблемой передачи параметров.

Знакомая незнакомка: арифметика

Итак, давайте на время отложим обсуждение подпрограмм и обратимся к более знакомой и прозаической задаче – вычислению арифметических выражений. Мы считаем своим долгом сразу же предупредить: знакомое отнюдь не означает простое. Арифметические выражения, с которыми все знакомы с начальной школы, кажутся простыми лишь на первый взгляд. Проблемы начинаются тогда, когда мы ставим перед собой задачу «научить» компьютер их вычислять. В общем, арифметические выражения – довольно крепкий орешек, в чем мы скоро убедимся. Так что не будем расслабляться – нам предстоит непростая работа, но она того стоит.

Будем считать, что арифметические выражения состоят только из чисел и знаков основных арифметических операций. Фактически арифметические выражения – это частный случай алгебраических выражений. В последних все или некоторые числа заменены буквами (или переменными). Алгебраическое выражение может быть вычислено, когда переменные получат числовые значения. Анализ и обработка арифметических и алгебраических выражений идентичны, а отличить арифметическое выражение от алгебраического легко из контекста. Обычно такое разделение несущественно, и мы будем говорить просто о выражениях.

В качестве знаков для операций мы будем использовать только четыре основных действия: сложение (+), вычитание (–), умножение (*) и деление (/).

Операции применяются к операндам (числам и переменным). Круглые скобки (и), как обычно, предназначены для группировки

операндов и указания последовательности действий при вычислении выражения. Вот несколько примеров, которые позволят нам вспомнить математику начальной школы (для выражений, чьи операнды – только числа, указаны результаты вычислений):

```
1+2      = 3
1+2*3    = 7
(1+2)*3  = 9
a+b
a+b*c
(a+b)*c
1+b/10
```

В дальнейшем будем предполагать, что скобочные структуры, имеющиеся в арифметическом выражении, составлены правильно, т. е. являются допустимыми (напомним, что задача проверки скобочных структур нами уже была ранее решена).

Наша задача теперь такова: вычислить заданное арифметическое выражение. Для арифметических выражений (первые три выражения из примера выше) это означает применить операции к операндам. Для алгебраических выражений это означает, что сначала нужно присвоить значения переменным; так получится арифметическое выражение, которое теперь может быть вычислено обычным способом.

Казалось бы, ничего сложного. Но давайте не будем торопиться; нам нужен алгоритм, который позволит вычислять арифметические выражения автоматически, т. е. на компьютере. И вот тут начинаются проблемы. Рассмотрим их по порядку.

Наш первый пример (т. е. 1+2), бесспорно, прост. Его структура элементарна настолько, насколько это возможно. Но уже второй пример не такой. И причина в том, что умножение должно быть выполнено прежде сложения. Принято говорить, что умножение имеет более высокий *приоритет*, чем сложение. То есть чем выше приоритет арифметической операции, тем раньше эта операция должна быть применена к своим операндам. Итак, второй пример должен быть вычислен так: умножить 2 на 3 (в результате получится 6), а уже после прибавить единицу. Если же мы хотели сделать иначе – сложить 1 и 2, а потом умножить результат на 3, то это необходимо описать явно, используя скобки (см. третий пример).

Приоритеты основных арифметических операций описываются простым правилом: в отсутствие скобок операции умножения (*) и деления (/) должны выполняться раньше операций сложения (+)

и вычитания (–). Иначе говоря, приоритет операций умножения и деления выше приоритета операций сложения и вычитания. Для изменения порядка выполнения следует использовать скобки. Сами скобки операциями не являются, но влияют на порядок вычислений.

Слова «больше», «меньше», «выше» и «ниже» правильно описывают смысл понятия приоритета, но им необходимо придать более точные значения.

Назначим каждой из операций некоторое числовое значение, которое будет характеризовать ее приоритет. Чем выше приоритет, тем большее числовое значение ему приписано. Вот соответствующая таблица приоритетов:

Лексема	Приоритет
+, –	5
*, /	10

Числовые значения приоритетов могут быть любыми; важно только, чтобы операции умножения и деления имели более высокое значение приоритета, чем операции сложения и вычитания. В этой небольшой таблице есть кое-какие «странности», но сейчас мы со всем разберемся по порядку.

Прежде всего из таблицы следует, что арифметические операции сложения и вычитания имеют одинаковый приоритет, равный 5; операции умножения и деления имеют одинаковый приоритет, равный 10. Так как, очевидно, 5 меньше 10, то операции сложения и вычитания должны выполняться после операций умножения и деления. Но нам нужно учесть еще и скобки. Скобки, конечно, не относятся к арифметическим операциям, но они влияют на порядок вычислений. Иногда скобки полезно рассматривать как некие псевдооперации и назначить им приоритет (меньший, чем у обычных операций), но мы не станем усложнять и учтем влияние скобок в алгоритме, о котором речь пойдет далее. Операнды (числа и переменные) играют в определенном смысле «пассивную» роль, поэтому они не включены в таблицу приоритетов.

Понятно, что эта таблица пока нисколько не помогла нам в понимании того, как вычислять арифметические выражения. В частности, неясным остается понятие «лексема». Это на самом деле простое понятие, но для этого нам придется коснуться терминологии из области трансляции языков программирования.

Наша задача, несмотря на ее «школьное происхождение», имеет самое непосредственное отношение к сложной и важной теме трансляции языков программирования, т. е. к преобразованию программ, написанных на каком-либо языке программирования, к виду, пригодному для исполнения на компьютере. Это может быть как объектный (двоичный) код, непосредственно понимаемый процессором, так и некоторое промежуточное представление. Впрочем, тема трансляции языков программирования настолько велика и в общем случае сложна, что мы будем ее касаться лишь в той степени, какая нам будет необходима.

Лексемы – это, грубо говоря, элементы арифметического или алгебраического выражения. Скобки, знаки операций, переменные и числа – суть элементы (или составляющие) выражения. В каком бы месте выражения не встретилась скобка, знак операции, переменная или число – все они будут классифицированы как лексемы определенных типов. Иначе говоря, говоря о лексемах, мы не делаем различий между числами 1, –88 или 0 – все это конкретные представители лексемы «число» (number). Совершенно аналогично переменные abc, value или counter относятся к лексеме «переменная» (variable). В последнем случае переменные abc, value и counter, конечно же, могут означать разные операнды, отличаться как составом входящих в них символов, так и длиной (соответственно, 1, 5 и 7), но все они являются лексемами одного типа, а именно – переменными.

Часто наряду с понятием «лексема» используется понятие «токен». Хотя, строго говоря, между ними и есть различия, но для нас здесь это не будет иметь значения.

Некоторые лексемы, как мы видели, очень просты и состоят из одного символа (скобки и знаки операций). Другие (переменные и числа) могут состоять из множества символов, но все числа относятся к лексеме «число», а переменные – к лексеме «переменная».

Проще говоря, лексема – это классификатор, описывающий, к какому классу относится элемент арифметического выражения.

Обратите внимание, что все переменные и все числа относятся к соответствующим лексемам. А вот скобки (и) мы разделим: каждой из них соответствует отдельная лексема. Сейчас, конечно, не ясно, почему мы так поступаем, но вскоре вы увидите причины подобного разделения.

Перед тем как приступать к анализу и вычислению выражений, нам придется еще раз обратиться к терминологии, используемой в об-

ласти трансляции языков программирования. Это тем более необходимо, что в последующих разделах эта терминология окажется нам полезной.

Привычный всем нам со школы способ записи выражений в виде a+b, в котором знак операции стоит *между* операндами, называется *инфиксной* (infix) формой. Он соответствует словесному выражению «а плюс b». Однако такой способ записи выражений не единственный и даже не самый лучший (почему – мы увидим чуть позже). Есть еще два способа записи выражений. Первый из них – *префиксный* (prefix), при котором знак операции стоит *перед* операндами, например (+ a b). Он соответствует словесному выражению «сложить a и b» (и, к слову, в обычной речи именно так мы и говорим). Преимущество префиксной формы записи сразу не очевидно, но префиксная форма позволяет записывать выражения в виде (+ a b c d) (в языке программирования Lisp), т. е. указывать одну операцию и любое количество операндов; в привычной инфиксной форме это же выражение мы должны были бы записать как a + b + c + d, т. е. использовать два «лишних» знака операции сложения.

Круглые скобки в предыдущем примере играют роль ограничителей – они указывают начало и окончание выражения.

Кроме того, префиксную форму в языках программирования обычно имеют подпрограммы; сначала указывается имя подпрограммы (оно играет роль операции), а за ним – список параметров (параметры играют роль операндов). Поэтому нет оснований утверждать, что префиксная форма – это нечто искусственное, неудобное и незнакомое.

Коль скоро мы уже видели и инфиксную, и префиксную формы, то методом исключения следует ожидать, что должна быть еще одна форма записи выражений, при которой знак операции стоит *после* операндов. Действительно, такая форма записи – *постфиксная* (postfix) форма – существует и, более того, представляет для нас особый интерес. При постфиксной форме записи предыдущий пример будет записан как a b +.

Постфиксная форма записи была предложена в 20-х годах XX века польским математиком Яном Лукасевичем. Научные интересы Лукасевича относились к математической логике, и ему нужен был способ записи логических уравнений, отличный от традиционного (в инфиксной форме), где так же требовалось учитывать приоритеты

логических операций и использовать скобки для изменения порядка их применения. Позже идеи Лукасевича были с успехом использованы разработчиками трансляторов языков программирования. Вероятно, сегодня программисты больше ценят значимость постфиксной формы записи, чем специалисты по математической логике – дисциплине, где эта форма впервые появилась. Постфиксная форма является одним из основных понятий при реализации языков программирования. Ни одна книга по трансляторам не обходится без того, чтобы обсудить способы перевода выражений в постфиксную запись. Впрочем, тут мы не только несколько забегаем вперед, но и вторгаемся в область, которая лежит несколько в стороне от наших задач.

Прежде чем приступать к описанию алгоритма трансляции выражений из инфиксной формы в постфиксную, приведем несколько примеров, снабдив их краткими комментариями. Рассмотрим выражение 2 + 3 * 4. Его значение, как легко видеть, равно 14. В постфиксной форме это выражение будет выглядеть так: 2 3 4 * + (как вывести эту форму, и составляет суть решения задачи данного раздела). А теперь посмотрим на выражение (2 + 3) * 4 (его значение равно 20). В постфиксной форме это выражение запишется так: 2 3 + 4 *.

Чуть раньше мы говорили, что постфиксная форма представляет для нас особый интерес. Это действительно так, и вот почему:

- в постфиксной форме сохраняется порядок следования операндов, хотя порядок следования операций может быть и изменен;
- в постфиксной форме нет скобок. Это говорит о том, что в постфиксной форме приоритеты всех операций одинаковы (забегая несколько вперед, можно сказать, что в постфиксной форме приоритеты операций вообще не имеют значения);
- выражение в постфиксной форме может быть вычислено путем однократного просмотра постфиксной записи слева направо (это утверждение пока мы примем на веру, т. к. еще не обсуждали, как именно производится само вычисление).

При всей необычности постфиксная форма записи обладает определенным изяществом: она строго линейна и не «прерывается» скобками, без которых в инфиксной форме обойтись в общем случае, увы, невозможно. Забегая вперед, скажем еще, что постфиксная форма идеально приспособлена для вычисления на компьютере.

А теперь мы опишем алгоритм трансляции выражений из инфиксной формы в постфиксную. Алгоритм будет изложен неформально (впрочем, сомнительно, что его запись в формализованном виде спо-

собствовала бы более ясному пониманию). После этого мы подробно разберем несколько примеров, которые, мы надеемся, полностью снимут все сомнения в корректности алгоритма и его практической ценности.

Опасаясь наскучить, все же вернемся еще раз к примеру a + b * c. Он, очевидно, вычисляется так: умножить b и c, затем сложить результат с a. Нас сейчас интересует слово «затем». Выражение считывается слева направо, и переменная a считывается первой. Хотя переменная a в записи выражения появляется раньше переменных b и c, используется она позже всех. Последней операцией является сложение. Эту операцию (т. е. сложение +) нужно на время «придержать» в запасе. И хотя мы применяем не строгую терминологию, эта терминология дает нам определенный намек на то, где должна храниться операция. Короче говоря, а не пригодится ли нам и тут наш уже хороший знакомый – стек? Надо попробовать...

Итак, нам потребуется стек. В стеке мы будем хранить скобки и символы операций. Этот стек будет называться *стеком операций*. Операндов выражений (т. е. чисел и переменных) в стеке операций не будет.

Алгоритм трансляции выражений

Входная строка, представляющая собой выражение (арифметическое или алгебраическое) в инфиксной форме, сканируется слева направо. По мере сканирования во входной строке выделяются лексемы. Если очередной выделенной лексемой является переменная или число (т. е. операнд), они сразу же записываются в выходную строку (изначально пустую). Если встречается лексема, соответствующая какому-либо знаку операции или скобке, то она анализируется по следующим правилам:

- если стек операций пуст, то операция проталкивается в стек; продолжение сканирования входной строки;
- если стек операций не пуст, то сравниваются приоритеты текущей (только что обнаруженной) операции и операции на вершине стека. Здесь возможны два случая: 1) приоритет операции на вершине стека ниже или равен приоритету текущей операции (скажем, на вершине стека лежит +, а текущая операция *), тогда текущая операция проталкивается в стек, и 2) приоритет операции на вершине стека выше приоритета текущей операции (например, на вершине стека лежит *, а текущая операция +), тогда выталкиваем из стека операций все

операции, чей приоритет выше приоритета текущей операции. После этого проталкиваем в стек текущую операцию и продолжаем сканирование входной строки;

- во входной строке встретилась левая скобка. Протолкнуть ее в стек операций и продолжить сканирование входной строки;
- во входной строке встретилась правая скобка. Отбросить эту скобку (т. е. не проталкивать ее в стек и не записывать в выходную строку) и выгрузить из стека операций в выходную строку все операции вплоть до левой скобки, но не включая ее. Отбросить левую скобку и продолжить сканирование входной строки;
- если достигнут конец входной строки, то вытолкнуть из стека операций все оставшиеся операции в выходную строку и остановиться.

Алгоритм из-за своей многословности выглядит несколько громоздко. Но давайте обратимся к примерам и разберем их шаг за шагом.

Пример 1: a + b

Стек операций	Инфиксная форма	Постфиксная форма
⊢	a + b	
⊢	+ b	a
⊢ +	b	a
⊢ +		a b
⊢		a b +

При трансляции выражений из инфиксной формы в постфиксную удобно записывать все этапы алгоритма трансляции в таблице, состоящей из трех колонок. В первой отражено состояние стека операций, во второй – еще не обработанная часть исходного выражения в инфиксной форме (входная строка), и, наконец, в третьей – выходная строка в постфиксной форме.

Что здесь происходит? Сначала считывается переменная a. Она, согласно алгоритму, сразу же попадает в выходную строку. Затем считывается операция сложения (+). Стек операций пуст, и эта операция немедленно проталкивается в стек. Потом считывается переменная b и сразу же записывается в выходную строку. Теперь вся входная строка прочитана, и в соответствии с последним пунктом алгоритма из стека операций выталкивается все его содержимое, т. е. в данном случае – одна операция +.

Этот пример настолько прост, что его можно было бы и не описывать так подробно. Но пока мы только учимся, и польза в детальном

воспроизведении работы алгоритма трансляции несомненна. Теперь рассмотрим чуть более сложный пример.

Пример 2: a + b – c

Стек операций	Инфиксная форма	Постфиксная форма
├	a + b - c	
├	+ b - c	a
├ +	b - c	a
├ +	- c	a b
├ + -	c	a b
├ + -		a b c
├ +		a b c -
├		a b c - +

Здесь приоритет у операции сложения такой же, как у операции вычитания, поэтому операции накапливаются в стеке. Обратите внимание на то, что порядок операндов в выходной строке совпадает с порядком операндов в исходном выражении, а вот порядок операций изменился. Теперь рассмотрим пример посложнее – с операциями разных приоритетов.

Пример 3: a * b + c

Стек операций	Инфиксная форма	Постфиксная форма
├	a * b + c	
├	* b + c	a
├ *	b + c	a
├ *	+ c	a b
├	+ c	a b *
├ +	c	a b *
├ +		a b * c
├		a b * c +

Тут определенно есть кое-что новенькое, и с этим следует аккуратно разобраться. Поначалу все идет, как и в предыдущих примерах: в стек проталкивается операция *, а в выходную строку записываются переменные a и b (строки с 1 по 4 таблицы). Затем считывается операция +. Приоритет операции сложения ниже приоритета операции на вершине стека (т. е. операции умножения). В соответствии с алгоритмом операция умножения * выталкивается из стека и записывается в выходную строку, а операция сложения +, напротив, запоминается в стеке (строки 5 и 6). Наконец, мы считываем остаток выражения в ин-

фиксной форме и формируем выходную строку в постфиксной форме.

Давайте изменим этот пример и посмотрим, как теперь будет идти трансляция.

Пример 4: a + b * c

Стек операций	Инфиксная форма	Постфиксная форма
├	a + b * c	
├	+ b * c	a
├ +	b * c	a
├ +	* c	a b
├ + *	c	a b
├ + *		a b c
├ +		a b c *
├		a b c * +

Полученная постфиксная форма уже совсем иная, чем в предыдущем примере. Давайте проанализируем, что тут происходит. Сначала идет знакомая нам уже последовательность действий (строки с 1 по 3). Потом мы встречаем операцию умножения (*) и сравниваем ее приоритет с приоритетом операции на вершине стека операций. Очевидно, что приоритет операции * выше приоритета операции +, и мы, согласно алгоритму, проталкиваем * в стек операции (строки 4 и 5 таблицы). Затем мы считываем переменную (c) и переносим ее в выходную строку. При этом мы доходим до конца входной строки и, согласно алгоритму трансляции, выталкиваем из стека операций все его элементы, начиная с вершины.

Во всех этих примерах мы не использовали скобок. Пора посмотреть, как будет работать алгоритм при наличии скобок.

Пример 5: (a + b) * c

Стек операций	Инфиксная форма	Постфиксная форма
├	(a + b) * c	
├ (	a + b) * c	
├ (	+ b) * c	a
├ (+	b) *c	a
├ (+	) * c	a b
├	* c	a b +
├ *	c	a b +
├ *		a b + c
├		a b + c *

Разберем этот пример подробнее, чем предыдущие, т. к. в нем отражается вся суть алгоритма трансляции.

Изначально стек операций пуст (строка 1). Затем (строка 2) мы считываем левую скобку. В соответствии с алгоритмом левая скобка проталкивается в стек операций. Читаем (строка 3) переменную и немедленно переносим ее в выходную строку.

Теперь мы дошли до первой настоящей арифметической операции (строка 4). Проталкиваем, как и предписано алгоритмом трансляции, эту операцию в стек. Затем (строка 5) идет переменная b, которая сразу же переносится в выходную строку. На этом этапе у нас осталась непрочитанной часть входной строки, а именно) * с. Читаем следующую лексему и обнаруживаем, что это правая скобка (строка 6). Обратимся к алгоритму трансляции. Из него следует, что нам нужно отбросить эту скобку и выгрузить в выходную строку из стека операций все операции, начиная с вершины стека и до левой скобки. Саму левую скобку мы не выгружаем, а просто отбрасываем как уже ненужную. Ну а дальше – уже знакомая нам по предыдущим примерам последовательность действий (строки с 7 по 9).

Давайте порассуждаем – что нового внесли в состояние стека скобки? Левая скобка служит своего рода барьером для накопления операций. Правая скобка служит для того, чтобы в нужный момент выгрузить из стека накопленные операции и снять барьер, образованный левой скобкой. Неформально говоря, левая скобка служит пружиной, которая сжимается под «весом» операторов, а правая – спусковым крючком, освобождающим пружину, после чего накопленные после левой скобки операторы выталкиваются в выходную строку. Сама левая скобка отбрасывается: она выполнила свою роль и больше не нужна.

Чтобы окончательно закрепить навыки использования алгоритма трансляции выражений из инфиксной формы в постфиксную, рассмотрим еще один пример, в котором мы не будем уже подробно описывать все шаги, оставив их в качестве упражнения.

Пример 6: (a + b *c)/(d – e)

Стек операций	Инфиксная форма	Постфиксная форма
├	(a+b*c)/(d-e)	
├ (	a+b*c)/(d-e)	
├ (	+b*c)/(d-e)	a
├ (+	b*c)/(d-e)	a
├ (+	*c)/(d-e)	a
├ (+ *	c)/(d-e)	a b
├ (+ *	)/(d-e)	a b c

Стек операций	Инфиксная форма	Постфиксная форма
├	/(d-e)	a b c * +
├ /	(d-e)	a b c * +
├ / (	d-e)	a b c * +
├ / (	-e)	a b c * + d
├ / (-	e)	a b c * + d
├ / (-	)	a b c * + d e
├		a b c * + d e - /

Забавно наблюдать, как по мере сокращения остатка исходного выражения в инфиксной форме растет выходная строка, представляющая собой выражение в постфиксной форме.

В заключение вернемся к таблице приоритетов и рассмотрим, как эта таблица может быть расширена, если необходимо ввести в нее новые операции. Пусть в дополнение к уже имеющимся операциям мы хотим обрабатывать операцию возведения в степень «^». Из школьной алгебры мы помним, что операция ^ должна выполняться раньше всех других операций, и поэтому таблица приоритетов приобретает следующий вид:

Лексема	Приоритет
+, -	5
*, /	10
^	15

Алгоритм трансляции остается тем же самым. Вот как будет обрабатываться выражение a + b * c ^ d (или a + (b * (c ^ d)) в форме с явным указанием порядка вычислений:

Стек операций	Инфиксная форма	Постфиксная форма
├	a+b*c^d	
├	+b*c^d	a
├ +	b*c^d	a
├ +	*c^d	a b
├ + *	c^d	a b
├ + *	^d	a b c
├ + * ^	d	a b c
├ + * ^		a b c d
├ + *		a b c d ^
├ +		a b c d ^ *
├		a b c d ^ * +

Еще раз мы видим, что в постфиксной форме записи выражения порядок операндов остался неизменным и порядок операций изменился в соответствии с их приоритетами.

Для чего потребовались такая большая работа и такой непростой алгоритм трансляции? Неужели лишь для того, чтобы показать возможность записи выражений в постфиксной форме? Конечно же нет. В следующем разделе мы поговорим о том, как вычисляются выражения в постфиксной форме. А сейчас, для закрепления навыков трансляции выражений из инфиксной формы в постфиксную, мы предлагаем самостоятельно придумать несколько примеров (чем больше, тем лучше) и поупражняться в их трансляции.

И последнее, но значительное замечание. Приведенный выше алгоритм трансляции не «умеет» обрабатывать ошибки в выражениях (мы не стали включать в него обработку ошибок, чтобы за этой обработкой не исчезла суть алгоритма трансляции выражений). Вот примеры таких ошибочных выражений: a+b/* и (a+)*b. Что нужно предпринять, если попытаться применить алгоритм трансляции к таким выражениям? Очевидно, что должна быть зафиксирована ошибка трансляции. Дополните алгоритм трансляции соответствующими возможностями и проверьте его на примерах.

Стек как вычислительное устройство

В этом небольшом разделе мы поговорим о том, что возможности стека не ограничиваются лишь хранением данных; стек может быть использован и в качестве вычислительного устройства. В следующем разделе мы опишем такое вычислительное устройство – небольшую стековую машину, а пока давайте вспомним, как мы использовали стек до сих пор.

Во всех задачах, которые мы решали прежде, стек использовался весьма однообразно: мы что-то проталкивали в него (скобки, адреса возвратов, операции) и что-то выталкивали. Стек исправно служил местом хранения информации в соответствии с дисциплиной LIFO. Никаких иных функций стек не выполнял. Но оказывается, что возможности стека не ограничиваются только лишь хранением информации; стек «способен» на нечто большее: он может выполнять функции вычислительного устройства. Сразу же оговоримся, что сам по себе стек не способен ни складывать, ни делить. Но он «может» делать нечто другое. Чтобы понять, как стек может быть использован в этом качестве, давайте вернемся к одному из примеров, который рассматривался нами в разделе о трансляции выражений.

Напомним, что там мы преобразовали выражение (a + b * c)/(d – e) в инфиксной форме к эквивалентному выражению в постфиксной форме a b c * – + d e – /.

Но тогда мы не обсуждали вопроса, для чего мы это делали. Всякое действие имеет смысл лишь тогда, когда приложенные усилия чем-то компенсируются. Усилия были немаленькими. Что же мы приобретаем, получив постфиксную форму?

Оказывается, постфиксная форма может быть очень элегантно вычислена с помощью стека. Только на этот раз в стеке будут храниться операнды операций, а не операции. Соответственно этому будем называть такой стек *стеком операндов*. Вычисление происходит по следующему очень простому алгоритму:

- если во входной строке, представляющей постфиксную запись выражения, встречается операнд, то он проталкивается в стек; количество элементов в стеке (или проще – глубина стека) увеличивается на один элемент;
- если во входной строке встречается операция, она выталкивает из стека операндов два верхних элемента, применяет к ним операцию и полученный результат проталкивает в стек (после чего глубина стека уменьшается на один элемент).

На примере, конечно, это видно лучше. Допустим, мы хотим вычислить последнее выражение в постфиксной форме. Для определенности положим a = 7, b = 4, c = 2, d = 11, e = 6. Традиционная инфиксная запись, после замены переменных их значениями, дает выражение (7 + 4 * 2)/(11 – 6), которое вычисляется в 3. А что даст постфиксная запись? Рассмотрим последовательно, как будет в этом случае идти процесс вычисления:

```
Стек                   Непрочитанная часть
|                      7 4 2 * + 11 6 - /
| 7                    4 2 * + 11 6 - /
| 7 4                  2 * + 11 6 - /
| 7 4 2                * + 11 6 - /
| 7 8                  + 11 6 - /
| 15                   11 6 - /
| 15 11                6 - /
| 15 11 6              - /
| 15 5                 /
| 3
```

По окончании вычисления в стеке остался результат, и именно такой, какой нужен. Итак, мы обнаружили еще одну полезную осо-

бенность стека, а именно: стек, оказывается, пригоден не только для трансляции выражений, но и для вычислений.

Мы, как и раньше, не включаем в алгоритм обработку ошибок и предполагаем, что стек данных содержит необходимое количество операндов. Если, например, перед выполнением любой из этих операций в стеке нет достаточного количества операндов (их, напомним, должно быть не меньше двух), то попытка выполнить операцию приведет к ошибке. Обычно такие проверки реализуются программным способом.

Эта особенность стека фундаментальна. Кроме того, она допускает очень простую реализацию. Практически все трансляторы языков программирования преобразуют выражения из инфиксной формы в постфиксную, после чего «передают» полученный результат стеку для вычисления.

Даже если бы возможности стека только этим и ограничивались, то уже это одно было бы немалым достижением, но, оказывается, стек может много больше. Он может служить основой программирования вообще, а не только в частном (хотя и важном) случае обработки выражений. Но для начала давайте порассуждаем, что должен содержать ориентированный на стек язык программирования, для того чтобы быть полезным.

Как солнце не без пятен, так и стек не без недостатков. Характерный для стека способ хранения и извлечения данных (LIFO) часто оказывается неудобным.

Многие структуры данных (например, массивы или записи) неразумно держать в стеке. Причина этого вполне очевидна: массивы состоят из большого количества элементов; записи часто включают в себя поля разных типов, а не только числа. Если, к примеру, массив состоит их 1000 элементов, то придется все их протолкнуть в стек. Это легко сделать (конечно, при условии, что стек способен вместить такое количество данных), но потом начинаются проблемы.

Принцип действия LIFO диктует строгий порядок доступа к данным. Если необходимые данные находятся где-то в глубине стека, то добраться к ним может оказаться затруднительным (как, например, мы можем получить доступ к 500-му элементу массива?). Для этого, начиная с вершины стека, придется вытолкнуть и где-то сохранить все «лишние» элементы, затем получить искомый элемент и, наконец, каким-то образом восстановить исходное состояние стека. Это долго, неэффективно и просто неудобно, но такая задача встречается очень

часто, и ее необходимо решать. Для этого в систему команд стековых машин вводятся специальные операции, позволяющие получить непосредственный доступ к элементам в глубине стека. Это, конечно, отступление от канонического определения стека, но практические соображения оказываются важнее. Кроме того, обычно глубину стека тем или иным способом ограничивают (тем более что память для стека «берется» из доступной памяти компьютера). При этом всегда существует вероятность переполнения стека, поэтому способ хранения в стеке больших объемов данных оказывается непригодным.

Другой способ хранения данных давно известен: использование памяти, не входящей в стек. Такая память называется *памятью с произвольным доступом* (random access memory, RAM). Именно так работают практически все известные языки программирования. Для ссылки на эту память используются адреса. Значит, язык программирования должен включать средства работы с памятью (чтение и запись).

Далее, чтобы язык был сколько-нибудь полезным, в нем должны быть предусмотрены средства управления последовательностью вычислений (в частности, проверка условий и циклы). Для этого в язык следует включить возможность команды передачи управления. Далее, в языке должны быть предусмотрены (как минимум) команды для выполнения основных арифметических (например, сложение и вычитание) и логических операций (конъюнкция, дизъюнкция, ...). Очень полезным средством разбиения больших программ на отдельные, относительно независимые части является механизм подпрограмм. В следующем разделе мы рассмотрим один из возможных вариантов решения этих проблем.

Стековая машина

Сейчас, следуя Филиппу Купману, мы опишем систему команд небольшой канонической (или обобщенной) стековой машины (generic stack machine). Эта машина очень проста, но, несмотря на это, она вполне пригодна для реализации практически любых алгоритмов.

Данные в стековой машине хранятся в трех областях памяти. Первая область – адресуемая *память произвольного доступа* (memory), подобная той, что имеется в любом компьютере. В этой памяти хранятся операции и данные. Вторая область – это *стек данных* (data stack, или сокращенно DS). В стеке данных хранятся операнды арифметических операций, адреса данных и адреса переходов. Третья область – *стек возвратов* (return stack, или сокращенно RS), предназна-

ченный для хранения адресов возврата из подпрограмм (но, как будет ясно из последующего, не только). Кроме того, имеется уже знакомый нам программный счетчик (program counter, или PC).

Система команд (instruction set) стековой машины включает в себя операции нескольких групп. Рассмотрим их по порядку, сопровождая краткими пояснениями и, где это представляется необходимым, примерами. Первая группа команд включает в себя операции манипулирования содержимым стека данных. Эти операции представлены в следующей таблице:

Операция	Стековая диаграмма
DUP	├ … a → ├ … a a
DROP	├ … a b → ├ … a
SWAP	├ … a b → ├ … b a
OVER	├ … a b → ├ … a b a

(многоточия в стековых диаграммах обозначают неуказанные элементы стека; их может быть много, а может быть – ни одного). Операция DUP (от duplicate) создает в стеке копию элемента на вершине стека, после чего стек становится больше (или глубже) на один элемент. Операция DROP удаляет (выбрасывает) элемент, находящийся на вершине стека. По существу, это синоним команды POP, о которой мы говорили в одном из предыдущих разделов. Теперь стек становится на один элемент меньше. Операция SWAP переставляет местами два верхних элемента стека; глубина стека не меняется. Наконец, операция OVER создает на вершине стека копию элемента, лежащего в стеке ниже его вершины, и глубина стека увеличивается на один элемент. Стековые операции не ограничиваются указанными, но перечисленные – наиболее важные и часто встречающиеся.

Вторая группа включает в себя арифметические и логические операции:

Операция	Стековая диаграмма
+	├ … a b → ├ … a+b
-	├ … a b → ├ … a-b
AND	├ … a b → ├ … a&a
OR	├ … a b → ├ … a|b
XOR	├ … a b → ├ … a^b

Операции этой группы очевидны сами по себе (важно только обратить внимание на то, что при вычитании вычитаемое находится на

вершине стека, а уменьшаемое – под ней). Все эти операции выполняются по одной схеме: операнды выталкиваются из стека, к ним применяется указанная операция, после чего результат проталкивается в стек; стек данных становится на один элемент меньше.

Третья группа операций предназначена для работы с памятью. В ней всего две операции:

Операция	Стековая диаграмма
!	⊦ ... n addr → ⊦ ...
@	⊦ ... addr → ⊦ ... n

Операция ! (сохранить, store) ожидает на вершине стека адрес памяти (memory), а под ним – данные, которые нужно сохранить. После выполнения этой операции данные (т. е. n) сохраняются по указанному адресу (addr). После этого и адрес, и данные из стека удаляются. Операция @ (извлечь, fetch) интерпретирует значение addr на вершине стека как адрес памяти и проталкивает в стек данных значение, хранящееся по этому адресу. Сам адрес памяти из стека удаляется.

Перед тем как рассматривать операции управления, рассмотрим небольшую группу, операции которой оказываются полезными для временного хранения данных без обращения к памяти:

Операция	Стековая диаграмма (DS)	Стековая диаграмма (RS)
>R	⊦ ... a → ⊦ ...	⊦ ... → ⊦ ... a
R>	⊦ ... → ⊦ ... a	⊦ ... a → ⊦ ...

Здесь задействованы оба стека (DS и RS). Операция >R выталкивает значение с вершины стека данных DS и проталкивает его в стек возвратов (RS). Операция R>, напротив, выталкивает значение с вершины стека возвратов (RS) и проталкивает его в стек данных (DS). Строго говоря, операции этой группы не являются обязательными (их легко заменить командами ! и @), но эти операции проще в использовании. Простота эта, однако, обманчива, т. к. по невнимательности или небрежности можно легко сделать недоступными адреса возвратов в RS. В определенном смысле операции >R и R> подобны скобкам: каждой операции >R должна соответствовать операция R>. Не следует злоупотреблять операциями этой группы, поскольку с их «помощью» можно привести оба стека в несогласованное состояние и полностью нарушить работу программы.

Следующая группа команд содержит операции управления. Эта группа немногочисленна, но без нее невозможно написать сколько-нибудь полезной программы:

Операция	Стековая диаграмма (DS)	Стековая диаграмма (RS)
IF	├ ... a → ├ ...	без изменений
CALL	├ ... addr → ├ ...	├ ... → ├ ... raddr
EXIT	без изменений	├ ... raddr → ├ ...

Операция IF выталкивает из стека данных значение на вершине стека и сравнивает его с 0. Если результат сравнения – истина (т. е. на вершине стека действительно был 0), то управление передается операции, чей адрес содержится в memory [PC]. Если же результат сравнения – ложь (т. е. на вершине стека данных был не 0), то выполняется следующая по порядку операция, т. е. операция по адресу memory [PC + 1]. Операция CALL осуществляет передачу управления подпрограмме по адресу на вершине стека. Адрес возврата (raddr), как и следует ожидать, проталкивается в стек возвратов. Наконец, операция EXIT осуществляет возврат из подпрограммы (PC = raddr).

Рассмотрим, наконец, последнюю группу, состоящую всего из одной операции:

Операция	Стековая диаграмма
LIT	├ ... → ├ ... n

Операция LIT, в сущности, является несколько «замаскированной» командой PUSH. Операнд, следующий за операцией, проталкивается в стек данных. Например, при выполнении последовательностей LIT 7 и LIT 13 стек данных будет изменяться следующим образом:

```
Стек данных          Операция
├ ...                LIT 7
├ ... 7              LIT 13
├ ... 7 13
```

Одним словом, операция LIT позволяет инициализировать вершину стека данных непосредственно в процессе исполнения программы. Это очень удобно, когда данных относительно мало.

Это несколько эскизное описание стековой машины дает тем не менее вполне адекватное представление о ней. В машине имеется практически все, что необходимо для написания программ: операции управления, операции работы со стеком и операции работы с па-

мятью. Однако представленная система команд, безусловно, не слишком удобна на практике.

Поясним это на примере операции @ (извлечение данных из памяти с последующей их загрузкой в стек данных). Перед тем как операция @ может быть исполнена, адрес памяти, в котором хранятся данные (addr, см. выше), должен уже находиться на вершине стека данных. Каким образом этот адрес туда попадет? Например, посредством операции LIT addr. Следовательно, на этапе составления программы мы должны заранее знать этот адрес. В небольших программах это определенно неудобство, а в программах чуть более сложных – уже проблема.

Будем честными: программа, написанная с использованием только перечисленных выше операций, становится сложной и плохо обозримой. Мы не столько решаем задачу, сколько поддерживаем стек данных в нужном состоянии. Относительно небольшие изменения в алгоритме могут привести к существенной переделке программы, что резко снижает продуктивность работы программиста и чревато возникновением ошибок, порой трудноуловимых.

Кроме того, структуры управления, очевидно, слишком бедны. В сущности, все, что у нас есть, – это операция IF, которая сравнивает значение на вершине стека данных с 0. А если нам нужно запрограммировать такое, например, условие: «передать управление, если число на вершине стека меньше 0»? Это возможно сделать и в текущей системе команд, но конструкция программы станет неочевидной, неуклюжей, и весьма быстро наступит разочарование в возможностях стековой машины.

В представленной системе команд нет удобных и привычных операций, таких, например, как изменение знака числа на вершине стека. Как можно было бы решить такую задачу? Вот примерный вариант:

```
Операция    Стек данных
            ├ ... n
LIT 0       ├ ... n  0
SWAP        ├ ... 0  n
-           ├ ... -n
```

Пусть на вершине стека данных находится число n. Необходимо изменить его знак. Это сделать просто: надо вычесть n из 0. Для этого проталкиваем в стек 0. Но пока вычитание производить нельзя, т. к. будет выполнено n–0, что, конечно, равно n. Нужно сначала поменять местами операнды, что позволяет сделать операция SWAP. Вот теперь можно вычитать (0–n), и на вершине стека будет находиться искомое значение –n.

Вместо одной операции целых три – слишком расточительно для такой простой задачи. Разумеется, можно оформить этот фрагмент в виде подпрограммы и обращаться к ней по мере надобности. А если задача будет сложнее? Скажем, нам может понадобиться найти абсолютное значение величины на вершине стека. Или выполнить умножение (с учетом знаков сомножителей). Обеих этих операций в исходной системе команд нет. Их придется программировать отдельно. Постепенно набор таких недостающих, но практически необходимых операций вырастает до внушительных размеров. Как следствие код программы, предназначенный для решения задачи, может легко «утонуть» под кодом подобных утилит.

Итак, в качестве первого приближения, или стартовой точки, каноническая стековая машина вполне подходит. Ее «возможностей», при должных затратах труда и времени, достаточно для реализации любых задач. Однако в качестве практического средства и полезного инструмента каноническая стековая машина слишком примитивна.

Какой вывод следует из всего этого? Неужели идея стековой машины потерпела фиаско и ни на что не пригодна, кроме как для решения самых простых задач? Вовсе нет. Стековая машина, если аккуратно и правильно дополнить ее возможности, может стать мощным инструментом. Во второй части книги мы опишем один из вариантов расширения канонической стековой машины, а в приложении приведем реализацию такой расширенной стековой машины (на языке программирования Java). Предлагаемые изменения будут простыми, но они позволяют существенно упростить нелегкую работу программиста, оставляя ему больше времени на размышления, чем на реализацию очередного паровоза.

Подпрограммы: передача параметров

После некоторого перерыва мы вновь возвращаемся к подпрограммам и к рекурсии. На этот раз нашей темой будет передача параметров.

Напомним, что подпрограммы предназначены для разделения большой программы на отдельные, относительно независимые части. Обычно подпрограмма предназначена для решения небольшой подзадачи, которую можно многократно вызывать из основной программы и других подпрограмм.

Отличительной особенностью всех предыдущих примеров подпрограмм было то, что в них не использовались параметры. Эти подпрограммы были просто наборами операций, выполнявших какие-то

действия, но без указания на то, какие данные при этом используются. Это, конечно, в известной мере помогло нам полностью сосредоточиться на механизмах управления подпрограммами. Теперь, когда эти механизмы в общих чертах стали понятными, настало время подумать и о данных. Говоря о данных, мы будем понимать прежде всего параметры (кроме параметров, есть еще и локальные переменные, но о них мы расскажем позже).

В связи с этим возникает один, немного странный вопрос: так ли уж необходимы параметры и можно ли обойтись без них? Ответ на этот вопрос положительный. Например, мы можем отвести для параметров часть адресного пространства, выделенного для программы, и обращаться к ним по адресам (обычно в языках низкого уровня, например ассемблерах) или по именам (в языках программирования высокого уровня). Когда параметры подпрограмм размещаются подобным образом, то они являются, по существу, частью *среды* исполнения программы, т. е. частью памяти, выделенной главной программе и ее подпрограммам.

Понятно, что в этом случае необходимость в параметрах автоматически отпадает: подпрограммам известно расположение (т. е. адреса) всех данных, в том числе и тех, которые выполняют роль параметров.

Но на этом все преимущества передачи параметров через среду и заканчиваются. Рассмотрим в качестве примера классический случай – найти максимальное из двух целочисленных значений. Вот фрагмент кода, который наверняка всем хорошо известен:

```
int a = 10;
int b = 20;
int m = 0;

if (a > b) m = a; else m = b;
```

Здесь мы видим две целочисленные переменные (a и b), из которых нужно выбрать одну – наибольшую – и присвоить ее значение третьей переменной m. В том или ином виде этот код часто встречается в программах сортировки и обработки массивов. Этот код так и «просится» быть оформленным в виде подпрограммы (как и прежде, мы используем язык программирования Java):

```
void max () {
    if (a > b)
        m = a;
    else
        m = b;
}
```

Очевидно, что все переменные, которыми манипулирует этот код, лежат вне подпрограммы; по отношению к подпрограмме они – *глобальные*. Перед вызовом этой подпрограммы ей должны быть известны адреса всех трех переменных: a, b и m. Попробуем улучшить эту подпрограмму. Сначала «избавимся» от переменной m:

```
int max () {
    if (a > b)
        return a;
    else
        return b;
}
```

Теперь наша подпрограмма «вырабатывает» максимальное значение и возвращает его в вызывающий код. Переменная m больше не нужна: любая другая целочисленная переменная может получить значение, возвращенное подпрограммой.

Теперь настала очередь переменных a и b. Наша подпрограмма «привязана» к внешним по отношению к ней переменным a и b и может работать только с ними (т. е. подпрограмме должны быть известны адреса памяти, отведенные переменным a и b). Если мы хотим, чтобы подпрограмма работала с любой парой целых чисел, а не только с теми, что хранятся в переменных a и b, ее следует еще раз переписать:

```
int max (int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}
```

Здесь мы вводим параметры, которые, чтобы отличить их от ранее используемых переменных a и b, обозначили как x и y. Теперь наша подпрограмма может находить наибольшее среди любых двух целых чисел. Она уже не зависит от конкретных чисел (a и b) и их расположения в памяти. Ее можно вызывать самыми разными способами, например max (10, 20), max (a, b), max (a, 100) или даже max (a, max (b, c)), т. е. найти максимальное из трех чисел. Иными словами, механизм параметров «отвязывает» подпрограммы от конкретных адресов памяти и делает подпрограммы независимыми (подпрограмма max теперь может стать частью библиотеки). Поэтому хотя теоретически без параметров можно и обойтись, но практически они настолько удобны и полезны, что не воспользоваться этими преимуществами было бы неразумно.

На этом со вступлением можно покончить и перейти к делу.

Для начала давайте приведем в некоторый порядок терминологию. Различают два типа параметров – формальные и фактические. *Формальные параметры* – это идентификаторы, которые указываются при определении подпрограммы. В нашем последнем примере это идентификаторы x и y (оба целого типа). *Фактические параметры* (часто используется термин *аргументы*) – это данные, для обработки которых подпрограмма предназначена. Например, в max (a, 100) фактические параметры – это целочисленная переменная a и число 100.

И формальные, и фактические параметры (аргументы) обычно записываются в виде списков, элементы которых разделены запятыми, и тогда о них говорят как о списках формальных и фактических параметров.

При вызове подпрограммы происходит сопоставление формальных и фактических параметров: первому формальному параметру соответствует первый фактический, второму формальному параметру соответствует второй фактически и т. д. Это т. н. позиционное сопоставление параметров.

Обычно требуется, чтобы количество формальных параметров совпадало с количеством фактических, но в некоторых языках программирования это требование не обязательно. Как в этих случаях происходит сопоставление формальных и фактических параметров при вызове подпрограммы, как и чем заменяются недостающие или лишние параметры – все это зависит от языка и должно быть отражено в документации к нему.

Теперь мы можем вернуться к главной теме этого раздела; нам нужно найти ответы на следующие вопросы: как в подпрограммы передаются параметры и как связываются между собой формальные и фактические параметры?

Ответы на эти вопросы отнюдь не очевидны; увы, но многие, даже весьма опытные, программисты этих ответов не знают и более того – даже не подозревают о том, что тут имеются сложности.

Часто программисты пользуются механизмом параметров как само собой разумеющимся, не задумываясь о том, как этот механизм устроен, какие проблемы при этом возникают и как эти проблемы решаются.

Сразу же мы должны предупредить: наше изложение не будет исчерпывающим, т. к. подробное изложение передачи параметров отно-

сится (опять и опять) к вопросам реализации языков программирования, а также к поддержке среды исполнения со стороны операционной системы. Этой теме посвящены соответствующие книги (некоторые из них приведены в библиографии). Мы лишь наметим основные контуры и дадим небольшое введение, отталкиваясь от этого, можно приступать к глубокому изучению и реализации параметров. Остаток этого раздела может быть опущен без ущерба для дальнейшего изложения, но если мы хотим понимать, как устроена внутренняя «кухня» языков программирования, без этого не обойтись. Кое в чем мы должны будем повториться, но это лучше, чем рисковать быть непонятым. Начнем с простых (т. е. нерекурсивных) подпрограмм и рассмотрим следующий фрагмент:

```
Адрес    Команда
...
         ; вызов подпрограммы
2000     JSR 2500
2001     ---
2002     ---
...
2500     ; начало подпрограммы
...
         ; сохранить результат (если он нужен)
...
         ; возврат из подпрограммы
2615     RET
...
         ; параметры
8000     10
8001     3
8002     0
...
```

Допустим, что, как и раньше, подпрограмма начинается с адреса 2500 и ожидает два параметра. Мы можем разместить эти параметры в ячейках памяти, скажем, в ячейках памяти с адресами 8000 и 8001.

Когда подпрограмме будет передано управление, ей понадобятся параметры. Их она сможет найти по адресам 8000 и 8001. Очевидно, что параметры, определенные таким образом, являются глобальными и доступны не только этой подпрограмме, но и всем другим, т. е. доступны отовсюду. Если при программировании на ассемблере с этим приходится как-то мириться, то в языках программирования высокого уровня это, бесспорно, плохое решение. Кроме того, адреса пара-

метров не могут быть изменены – подпрограмма будет искать нужные ей параметры именно по указанным адресам.

Итак, передача параметров через выделенные ячейки памяти (или, что то же самое, через глобальные переменные) – плохая практика. Случаи, когда глобальные переменные действительно полезны, весьма немногочисленны, и всякий раз их использование должно быть обоснованным.

Где еще могут располагаться параметры подпрограмм? Их можно, например, располагать в теле самой подпрограммы, скажем, непосредственно в начале или непосредственно возле команды возврата RET. Но это фактически ничего не меняет – параметры как были глобальными, так ими и остались. Ведь в ассемблере мы можем легко передать управление подпрограмме, не используя операцию JSR (в результате чего рискуем полностью нарушить ее работу). Нам надо, чтобы подпрограмма не зависела от расположения параметров, а просто получала их в момент вызова. Где же еще могут находиться параметры?

Хотя мы не будем подробно останавливаться на способах передачи параметров, но упомянуть об этом будет нелишним. В основном используются два способа передачи параметров в подпрограммы – по значению и по ссылке.

При передаче параметров по значению подпрограмме передаются не сами данные, которые подпрограмма будет обрабатывать, а копии этих данных. При этом адреса данных подпрограмме неизвестны и, следовательно, «оригинальные» значения подпрограмма изменить (или, лучше сказать, испортить) не может.

При передаче параметров по ссылке подпрограмме передаются адреса данных, которые подпрограмма будет обрабатывать. Поскольку при этом подпрограмме известны адреса данных, переданные по ссылке, то подпрограмма имеет доступ к ним и может их изменить (не следует отождествлять передачу параметров по ссылке с глобальными переменными; хотя в обоих случаях подпрограмма и работает с адресами памяти, но не ко всем адресам подпрограмма может обращаться, а лишь к тем, что ей были явно переданы в качестве параметров в момент вызова). Какие способы передачи параметров допустимы, определяется в основном реализацией языка.

Различные языки программирования накладывают различные ограничения на способы передачи параметров. Например, в Java примитивные данные передаются только по значению, в то время как в C

или в C++ их можно передавать и по значению, и по ссылке. Составные типы данных (массивы, записи, объекты) передаются, как правило, по ссылке.

Так где же следует располагать параметры? Ответить на этот вопрос непросто. Начнем издалека...

Давайте вспомним, как устроен механизм вызова рекурсивных подпрограмм. Для того чтобы обеспечить правильный возврат из цепочки рекурсивных вызовов, мы использовали стек. В каждый момент времени выполняется только один из рекурсивных вызовов подпрограммы, а именно последний. Если мы находимся внутри рекурсивного вызова, все более ранние рекурсивные вызовы приостанавливаются. Это, как мы помним, называется вложенностью подпрограмм. Но ведь каждая выполняющаяся подпрограмма работает со своим набором параметров! Посмотрим на следующий пример:

```
int f = factorial (4);                      // Точка P

int factorial (int n) {
    if (n <= 0) return 1;
    return (n * factorial (n - 1));         // Точка S
  }
```

(рекурсивный вызов подчеркнут; о точках P и S в комментариях чуть ниже). Вычисление факториала при n = 4 происходит следующим образом:

```
factorial(4)=
   4*factorial(3)=
      4*3*factorial(2)=
         4*3*2*factorial(1)=
            4*3*2*1*factorial(0)=4*3*2*1*1=24
```

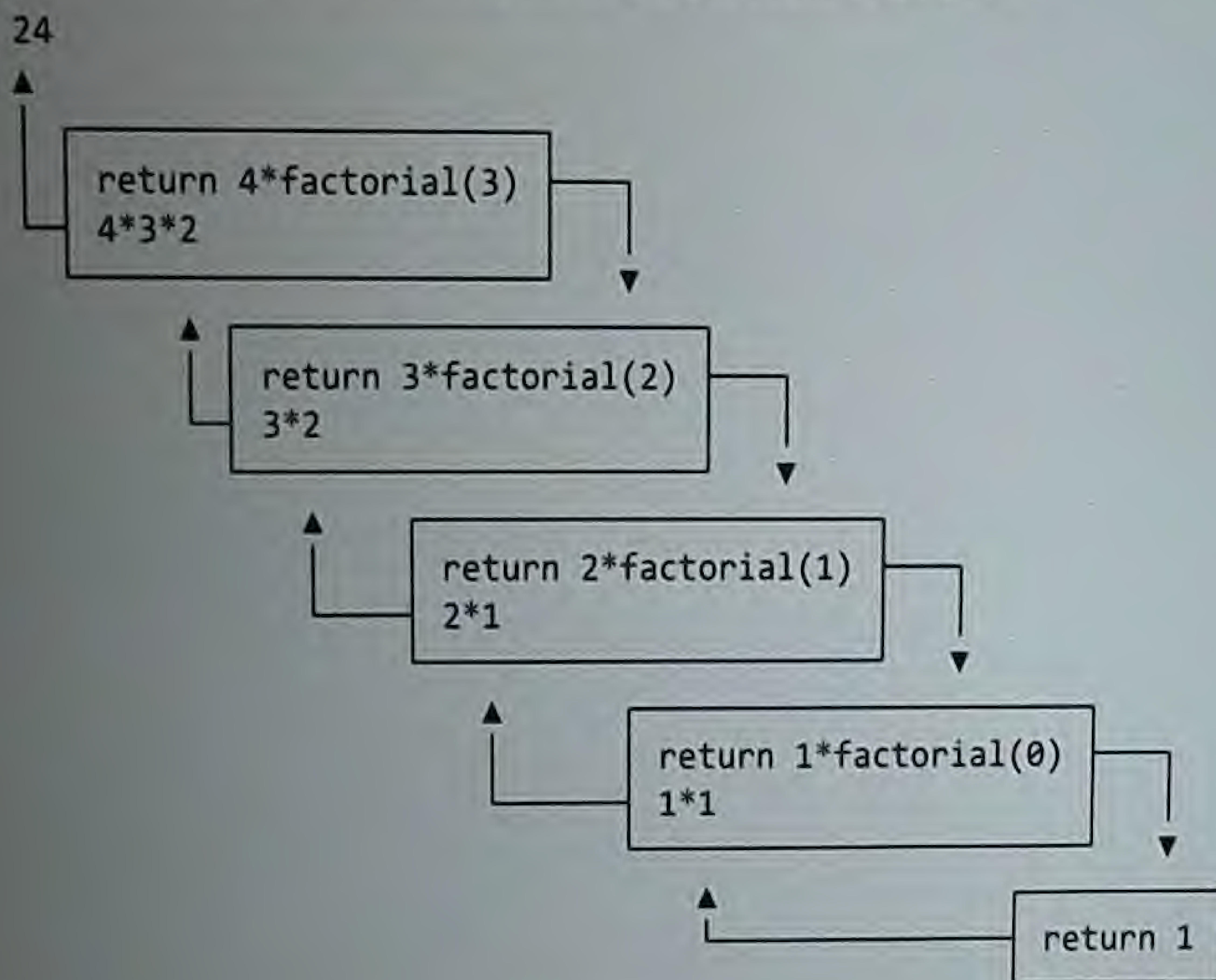
Здесь видно несколько рекурсивных вызовов, отличающихся друг от друга значением фактического параметра.

Еще раз посмотрим на пример. В нем выделены две точки: P означает главную программу, S – точку рекурсивного вызова. Вот как будет меняться содержимое стека возвратов при вычислении факториала:

```
|                  перед вызовом подпрограммы
|- P                  вызов factorial (4)
|- P S                  вызов factorial (3)
|- P S S                  вызов factorial (2)
|- P S S S                  вызов factorial (1)
```

```
├ P S S S S                  вызов factorial (0)
├ P S S S S                  return 1
├ P S S S                  return 1*1
├ P S S                  return 2*1*1
├ P S                  return 3*2*1*1
├ P                  return 4*3*2*1*1
├                  return 24; возобновление работы главной программы
```

И наконец, то же самое, но в графической форме:



(стрелки, выходящие из прямоугольных блоков направо и вниз, – это рекурсивные вызовы; стрелки, выходящие из прямоугольных блоков налево и вверх, – возвраты из рекурсивных вызовов).

Мы привели три способа изображения процесса выполнения рекурсивной подпрограммы, и во всех случаях мы хотели подчеркнуть следующий факт, имеющий фундаментальное значение: *при каждом рекурсивном вызове* подпрограммы factorial ей передается параметр на единицу меньше предыдущего, и так до тех пор, пока этот параметр не станет равным 0 (условие завершения рекурсии).

Поскольку это очень важный и тонкий момент, мы повторимся еще раз: параметр передается при каждом рекурсивном вызове! Или иначе: каждый рекурсивный вызов сопровождается передачей соот-

ветствующего параметра, и уже поэтому каждый рекурсивный вызов отличается от других (а именно значением параметра).

Если в рекурсивной подпрограмме предусмотрено несколько параметров, то каждый рекурсивный вызов отличается от других значением, по крайней мере, одного из параметров. Почему это важно? Вспомним об условии завершения рекурсии: рекурсивные вызовы должны завершаться при выполнении определенного условия. Если все рекурсивные вызовы неотличимы друг от друга, как можно ожидать, что они завершатся? Никак.

Разумеется, могут быть и подпрограммы без параметров; в этом случае условие завершения рекурсии должно находиться вне подпрограммы.

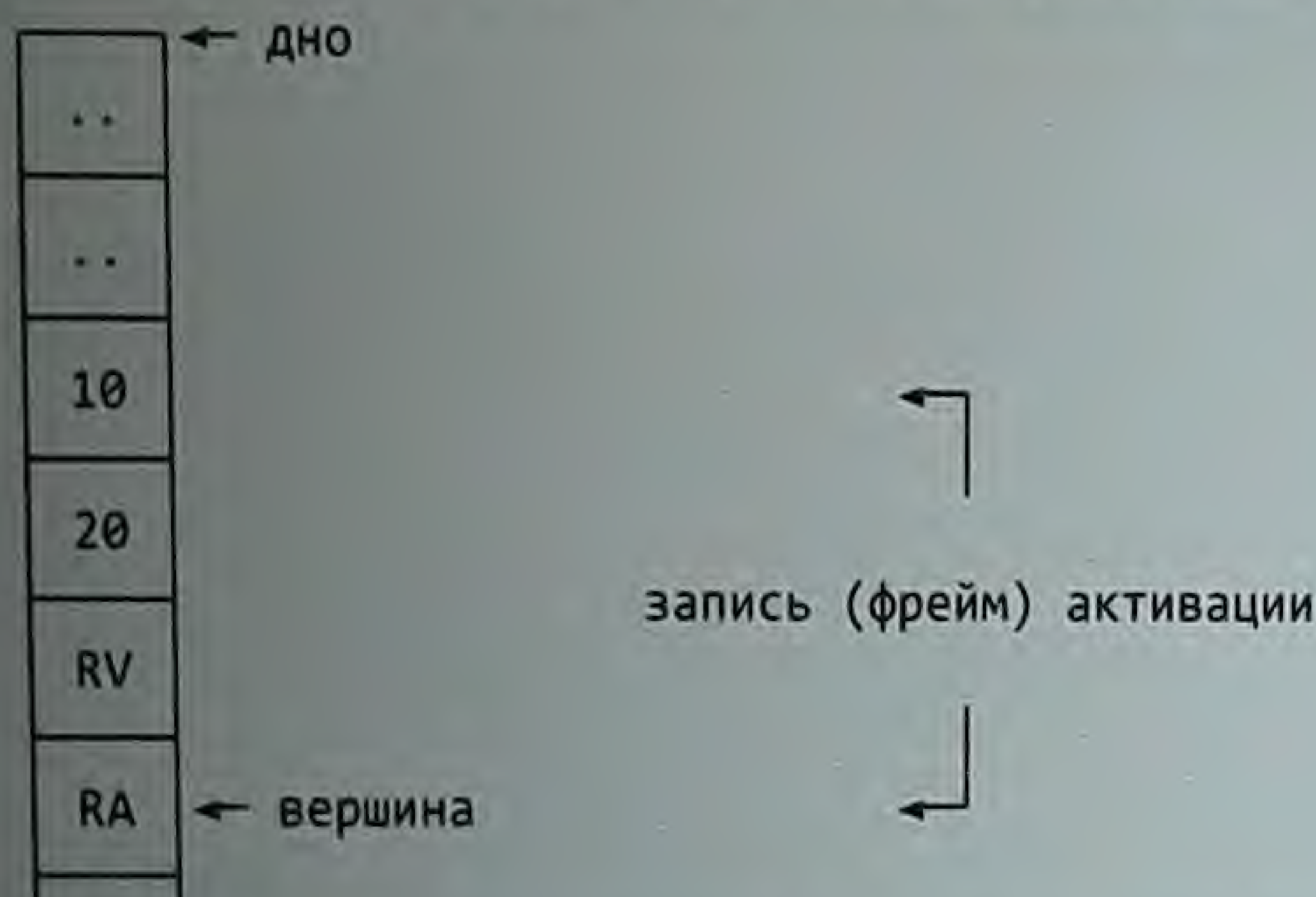
Коль скоро каждый рекурсивный вызов сопровождается проталкиванием адреса возврата в стек возвратов, то почему бы не проталкивать туда же (т. е. в стек возвратов) и параметр? Это сильно смахивает на провокацию, но давайте не будем торопиться – кто знает, может быть, в этом что-то есть.

Посмотрим на проблему с другой точки зрения: когда возникает и отпадает необходимость в параметрах, т. е. в какие моменты времени для них должна выделяться и освобождаться память? Ответ уже должен быть очевиден: выделение памяти для параметров производится тогда, когда управление передается подпрограмме. До этого момента они *еще не нужны*. Соответственно, освобождение памяти, выделенной для параметров, должно производиться тогда, когда подпрограмма завершается. После этого момента они *уже ни к чему*. Вход в подпрограмму сопровождается проталкиванием в стек возвратов адреса возврата из подпрограммы. Так давайте протолкнем в этот же стек и параметры. Более того, если уж «рисковать», то по-настоящему: протолкнем в стек возвратов и возвращаемое значение (если, разумеется, подпрограмма его вырабатывает)!

Посмотрим, как это может выглядеть на деле. Приведем еще раз ранее уже знакомую нам подпрограмму нахождения наибольшего из двух чисел:

```
int max (int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}
```

Пусть обращение к этой подпрограмме, например, такое: max (10, 20). Вот как может выглядеть стек возвратов при входе в эту подпрограмму (мы воспользуемся графическим представлением стека; напомним, что дно стека располагается в старших адресах памяти, а стек растет сверху вниз, т. е. в сторону младших адресов):



Это называется *записью активации*, или *фреймом активации*.

При вызове подпрограммы происходит следующее: сначала в стек проталкиваются фактические параметры или аргументы (10 и 20), затем в стек проталкивается возвращаемое значение (RV от return value), которое на этом этапе пока неизвестно, а будет вычислено позже, и, наконец, адрес возврата (RA от return address). Заметьте: теперь мы работаем только со стеком; основная память вообще не используется.

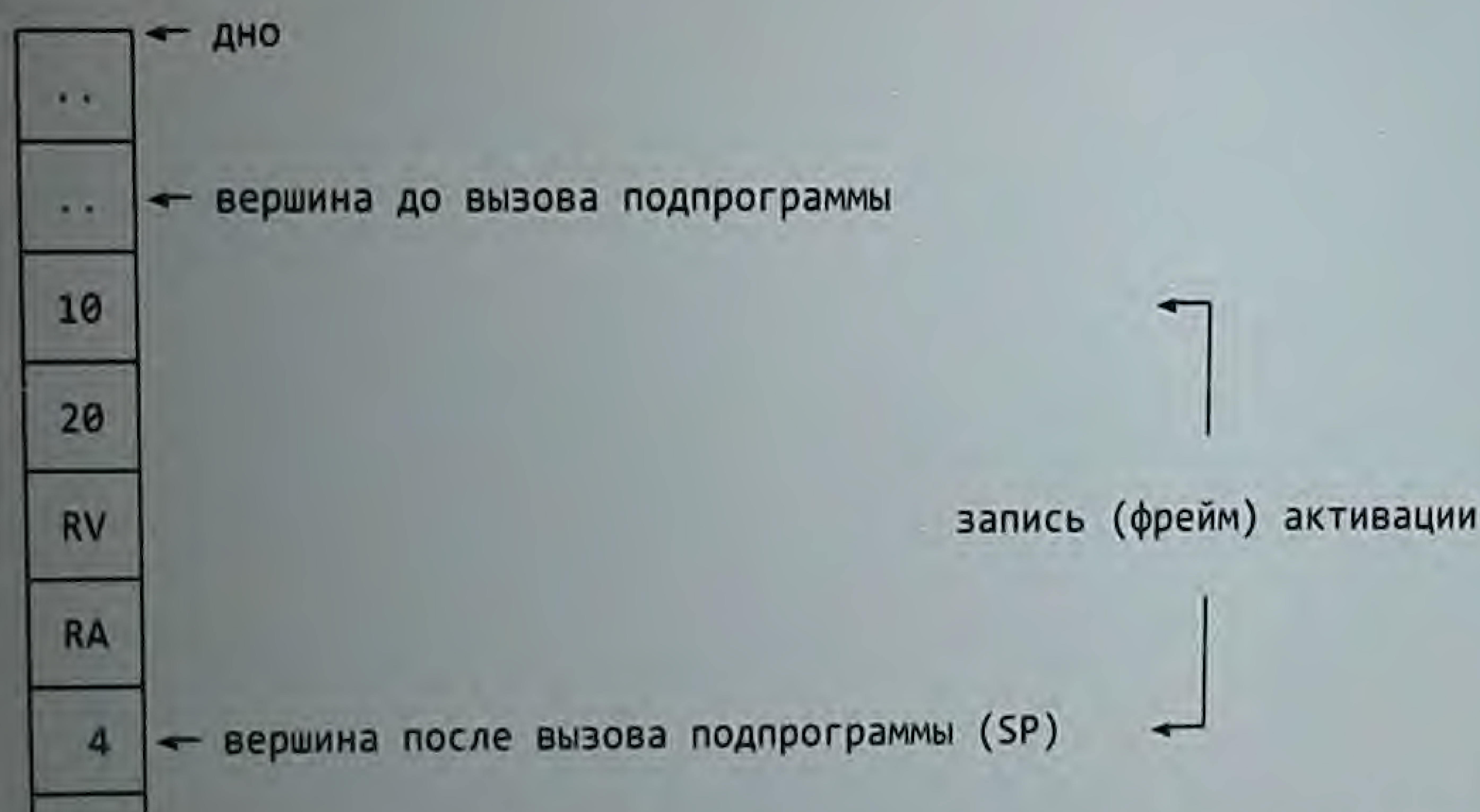
Порядок проталкивания в стек параметров, адреса возврата и возвращаемого значения может быть и иным. В этом вопросе нет ограничений, а выбор порядка определяется системой команд процессора и программистом, который реализует данный механизм.

Теперь подпрограмма начинает выполняться. После того как работа подпрограммы будет завершена, произойдет возврат из нее в вызывающую программу. Но позвольте, а как же возвращаемое значение и параметры? Они как лежали в стеке, так в нем и остались. Совершенно очевидно, что по завершении подпрограммы их из стека нужно удалить и вернуть стек в состояние, в котором он находился до вызова подпрограммы. Но как? Это определенно проблема. Но все не так плохо, как может показаться на первый взгляд.

Еще раз посмотрим на подпрограмму max. Подпрограмма ожидает два параметра (x и y) и возвращает целое значение, т. е. всего три ве-

личины. Кроме того, учтем еще и адрес возврата. Все это известно до того, как подпрограмма будет вызвана. Следовательно, нам точно известно количество элементов, которое надо будет протолкнуть в стек при вызове подпрограммы (а именно 3 + 1 = 4). Еще раз посмотрим на стек. В нем находятся четыре элемента: адрес возврата (RA), возвращаемое значение (RV) и два параметра, т. е. как раз столько, сколько мы только что посчитали. Следовательно, по окончании работы подпрограммы мы уже знаем, какое количество элементов должно быть удалено из стека.

Чтобы учесть последнее обстоятельство, надо где-то запомнить, сколько ячеек стека при вызове подпрограммы было нами фактически «использовано». Повторяем, что это заранее известная величина. Следовательно, мы можем протолкнуть в стек это число. Теперь стек станет таким:



Давайте еще раз перечислим, что находится в стеке непосредственно перед выполнением подпрограммы (в обратном направлении, т. е. начиная от вершины стека в сторону его дна):

- общее количество элементов, сохраненных в стеке (4);
- адрес возврата (RA);
- возвращаемое значение (RV);
- второй фактический параметр (20);
- первый фактический параметр (10).

Подпрограмма начинает исполняться. Когда ей потребуются фактические параметры, она их «найдет» в стеке на позициях SP + 3 (второй параметр) и SP + 4 (первый параметр).

Мы вновь напоминаем, что стек растет в сторону меньших адресов памяти, поэтому для доступа к более «ранним» элементам стека значение SP нужно увеличивать. Здесь надо быть внимательным, в противном случае неизбежны грубые ошибки и даже потеря данных.

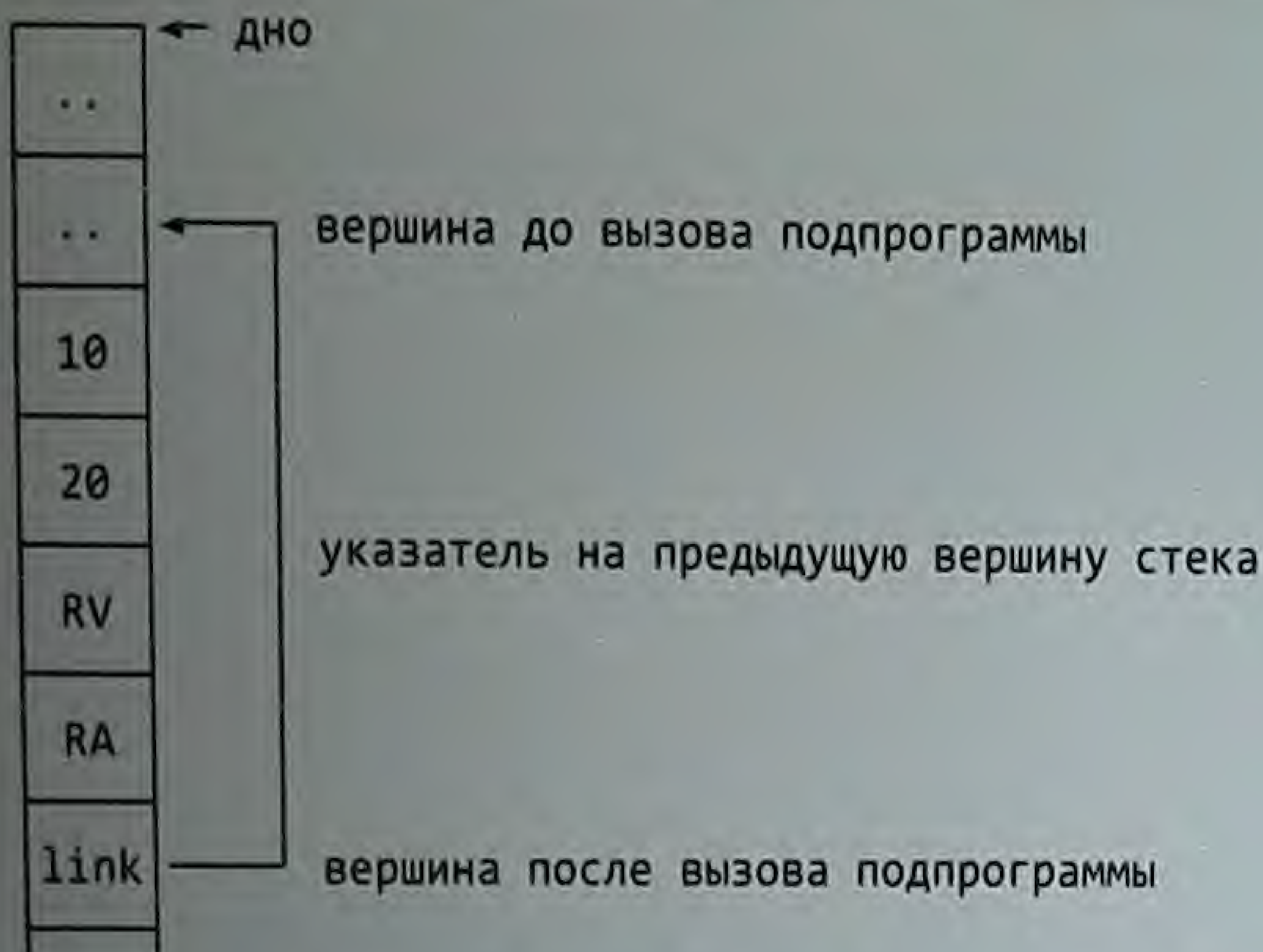
Когда подпрограмма будет готова вернуть управление вызывающей программе, она должна будет прочитать значение возвращаемого значения (SP + 2) и адрес возврата (SP + 1). Но это еще не все. По окончании работы подпрограммы необходимо вернуть стек в состояние, в котором он находился до вызова подпрограммы. Это сделать очень просто: нужно всего лишь скорректировать значение указателя стека SP = SP + [SP] + 1. Почему именно так? Да очень просто:

- сначала указатель стека корректируется на число элементов, которые в нем были размещены. Это число равно 4, и получить его можно, прочитав значение элемента на вершине стека, а это – не что иное, как [SP];
- нужно не забыть, что эта четверка тоже занимает ячейку памяти и ее нужно учесть; поэтому значение SP необходимо скорректировать еще на единицу.

После этого стек возвращается в состояние до вызова подпрограммы. Сложно? Да, кое-какие проблемы есть, но они носят чисто технический характер и решаются при реализации языка программирования; нужны всего лишь аккуратность и тщательный подсчет.

В системе команд некоторых процессоров имеются специальные операции для резервирования и освобождения указанного количества элементов стека. Операция резервирования обычно называется ENTER n, где n – количество элементов, на которое нужно увеличить указатель стека. Операция освобождения стека называется LEAVE n, где n – количество элементов, на которое нужно уменьшить указатель стека.

Вместо хранения числа элементов чаще используется не счетчик, а указатель на вершину стека до вызова подпрограммы (т. е. на предыдущую вершину стека):



Здесь link – это указатель на предыдущую вершину стека.

Кстати, заметим, что, как правило, никакие элементы в стеке не удаляются. Это совершенно излишне – ведь к ним все равно нет доступа, так что не стоит тратить времени на это. Действительно, достаточно изменить значение указателя стека SP (обычно говорят проще – «сдвинуть указатель стека»).

Мы помним, что стек – это структура данных, доступ к элементам которой осуществляется только в одном месте: на вершине стека. Предыдущие пояснения могут заставить в этом усомниться, и не без оснований. Для доступа к параметрам нам пришлось выбирать их параметры не на вершине стека (т. е. не там, куда указывал SP), а глубже – в позициях SP + 3 и SP + 4. Как же так?

А вот так! Теоретически, конечно, этого делать, может, и не следовало бы, но практически это полезный прием, и пренебрегать им ради сомнительной «чистоты» будет неразумным. Да, мы обращаемся со стеком не совсем так, как предписано его формальным определением. Но это полезно, удобно и практично. Поэтому мы так и поступили.

Описанный только что прием размещения в стеке дополнительных данных (помимо адреса возврата) может быть реализован многими способами. В нашу задачу не входит рассмотрение всех вариантов и тем более того, как это реализуется практически. Для этого лучше обратиться к соответствующей литературе. Нам же важно обратить внимание на то, что это возможно (и, к слову, действительно используется на практике).

В начале этого раздела мы упоминали, что при обращении к подпрограммам в стеке можно сохранять и локальные переменные, т. е. локальные переменные, чья область видимости ограничивается только подпрограммой. Память для локальных переменных выделяется и освобождается точно так же, как и для параметров, т. е. в момент вызова подпрограммы и в момент ее завершения. Но тут надо иметь в виду три обстоятельства.

Первое: если мы начнем проталкивать в стек еще и локальные переменные (скажем, непосредственно до или после фактических параметров), то возникнет сложность – как отличить одни от других? Следовательно, в стеке нужно будет как-то указать, где в нем расположены локальные переменные, а где – параметры. Это нетрудно сделать. Например, добавив в стек символ-разделитель или отдельный счетчик для параметров. Можно добавить в стек указатели: один – на начало секции параметров, второй – на начало секции локальных переменных. Конечно, структура записи активации станет сложнее, но при аккуратной реализации мы всегда сможем разобрать ее на составные части: ссылка на предыдущую запись активации, адрес возврата, возвращаемое значение, список параметров, список локальных переменных.

Второе: в блочно-ориентированных языках, в которых подпрограммы могут вкладываться друг в друга (например, в Pascal), вложенные подпрограммы могут обращаться к локальным переменным, принадлежащим «родительской» подпрограмме. Копировать их в запись активации вложенной подпрограммы бессмысленно – их значения будут утеряны после возврата из вложенной подпрограммы. Нужно организовать к ним доступ (обычно посредством ссылок из записей активации вложенных подпрограмм).

И наконец, третье. Во всех наших примерах мы явно и неявно предполагали, что и параметры, и локальные переменные относятся к одному типу данных – к целым числам. На практике обычно это не так; в качестве параметров и локальных переменных могут выступать строки, действительные числа, логические значения и проч. Как в одном стеке разместить столь разнородные данные? Конечно, и в этом случае решение есть (и не одно), но обсуждение этого увело бы нас слишком далеко и почти ничего не добавило к главной теме этого раздела – передаче параметров и локальных переменных. Обо всем этом лучше почитать в специальной литературе (см. библиографию).

На этом мы завершаем раздел, посвященный передаче параметров в подпрограммы. Он, вероятно, оказался самым сложным из всех разделов. Признаемся честно, что мы затронули только самые простые

вопросы и предложили самые очевидные варианты решений. На самом деле управление подпрограммами – тема огромная и относится больше к вопросам трансляции языков программирования и устройству операционных систем.

Стек и грамматики

В этом (и последнем) разделе первой части книги мы рассмотрим один интересный метод решения задач определенного класса (распознавание цепочек символов), а также расширение этого метода путем присоединения стека. Этот метод (вместе с его расширением) позволяет, в частности, по-новому решать задачи анализа скобочных структур и трансляции выражений, рассмотренных ранее.

Может показаться, что коль скоро задача решена, то искать другие способы ее решения бессмысленно; к чему тратить время и силы, пытаясь по-новому решить то, для чего решение уже есть?

Это заблуждение, и, к сожалению, весьма распространенное. Новые способы не просто по-другому решают уже решенные задачи. Новые способы открывают новые возможности, а это гораздо важнее. Может оказаться, что эти способы решают старые задачи проще и эффективнее, но это, пожалуй, не главное. Новые способы решают то, что раньше решалось плохо, частично или решение чего было неизвестно.

Обсуждая предыдущие задачи, мы говорили о том, что многие из них относятся к области трансляции языков программирования. Пора, наконец, хотя бы вкратце рассказать о том, что это такое.

При составлении программы, решающей какую-либо задачу, мы обычно пользуемся тем или иным языком программирования: Java, C, C++, C#, Python, PHP и т. д.; такая запись называется *исходным текстом*, или *исходным кодом*.

В те далекие времена, когда языков программирования еще не было, программы либо набирались на специальных штекерных панелях, подобных тем, что изредка встречаются на древних телефонных коммутаторах, либо записывались непосредственно в двоичных кодах машины.

Перед тем как эта программа будет исполнена, ее необходимо преобразовать к форме, «понятной» компьютеру, – обычно в последовательности из 0 и 1 (но не всегда; часто целью такого преобразования является некая промежуточная форма, которая может быть выполнена на подходящей машиной, т. е. программой-интерпретатором).

Эта форма обычно называется *объектным кодом*. Обе формы (исходная и объектная) эквивалентны в том смысле, что они представляют одну и ту же задачу. Процесс преобразования программы из исходного кода в объектный и называется *трансляцией*, т. е. *переводом*. Такое преобразование осуществляют специальные программы – *трансляторы* (или *компиляторы*).

В каком-то смысле подобный перевод аналогичен переводу с одного естественного языка на другой (скажем, с английского на немецкий). Правда, в естественных языках качество перевода зависит от таланта и квалификации переводчика, а потому один и тот же оригинальный текст переводится разными переводчиками по-разному. Это вполне допустимо, но только при условии, что не искажается смысл оригинала. Но если это не так, то перевод не просто плох: он может ввести в заблуждение и даже навредить (например: какими будут последствия ошибки, допущенной при переводе рецептуры лекарства; что может произойти, если переводчик вместо микровольт написал киловольты?).

В программировании такая ситуация должна быть исключена – перевод должен быть гарантированно точным, т. е. таким, чтобы полученный объектный код, который будет исполняться, полностью соответствовал задуманному исходному коду. Почти никогда нет возможности переводить исходный код разными способами, смотреть, что из этого получится, и выбирать подходящий вариант: надо сразу переводить правильно. Ведь программы контролируют множество важных процессов (производство, вооружения, системы жизнеобеспечения, распределение энергии, космос, связь, транспорт, финансы), и мы должны быть уверены, что объектный код в точности соответствует тому, что было задумано программистом.

В программировании говорить о смысле программ следует осмотрительно: это понятие плохо формализуемо. Попробуйте, например, дать устраивающие всех определения слов «любовь» или «надежда». Как определить понятие «между»? Помните, что определяемое не должно использоваться в теле определения, иначе получится порочный круг! Аналогичные затруднения возникают и со словом «смысл».

В сравнении с естественным языком выразительные средства всех языков программирования чрезвычайно бедны. В первом мы располагаем не только большим словарным запасом, но и большим числом способов передачи мыслей, понятий или сообщений. Естественный язык очень часто неоднозначен, и для понимания смысла нужен контекст. Но зато благодаря этим «недостаткам» естественного языка, его из-

быточности и исключениям, у нас есть литература и поэзия. Так что отсутствие абсолютной строгости порой совсем не плохо.

В области трансляции языков программирования имеются свой, и весьма развитый, математический аппарат (впрочем, вполне доступный всякому неглупому человеку), своя терминология и свои способы решения задач. Мы не станем в этой книге даже пытаться сделать невозможное, а поэтому ограничимся лишь начальными сведениями, достаточными для того, чтобы понять последующие примеры.

Для начала нам понадобится одно важное понятие – *конечный автомат* (коротко КА), а точнее – детерминированный КА.

Есть еще и недетерминированные КА, и они тоже важны, но мы ограничимся здесь лишь детерминированными КА.

Несмотря на маловразумительное (поначалу) название, практически КА оказывается весьма простым и, главное, удивительно полезным инструментом. Основное назначение КА – распознавать цепочки символов. Некоторые из таких цепочек считаются правильно сформированными (или допустимыми); остальные цепочки КА будет отвергать.

Все это, конечно, не более чем слова, поэтому рассмотрим простой пример: дана цепочка из 0 и 1, написанных в любом порядке; цепочка допускается, если в ней содержится нечетное количество единиц. Очевидное решение: завести счетчик для подсчета количества единиц с начальным значением 0 (поскольку в начале процедуры распознавания еще ни один символ не был прочитан). Всякий раз, когда в цепочке встречается единица, значение счетчика увеличивается на 1. Если вся цепочка прочитана, а значение счетчика нечетно, то эта цепочка допускается, иначе – отвергается. Но мы решим эту задачу иначе, с использование КА.

КА для решения этой задачи задается очень просто. Каждый новый символ изменяет состояние КА. Новое состояние зависит от текущего входного символа и текущего состояния и может быть задано *функцией перехода* (обозначим ее буквой Δ):

```
Δ (ЧЕТ,   0) → ЧЕТ
Δ (НЕЧЕТ, 0) → НЕЧЕТ
Δ (ЧЕТ,   1) → НЕЧЕТ
Δ (НЕЧЕТ, 1) → ЧЕТ
```

То есть если текущее состояние КА – это ЧЕТ и очередной символ – это 0, то КА остается в том же состоянии, и т. д. Начальное со-

стояние КА – это, конечно, состояние ЧЕТ (прочитано нуль символов, а нуль – четное число). Если КА останавливается в состоянии НЕЧЕТ, то цепочка допускается. Такое состояние называется допускающим. Если КА останавливается в состоянии ЧЕТ, то такое состояние называется отвергающим. Рассмотрим цепочку 10110101. В ней содержатся 5 единиц. Посмотрим, что нам «скажет» КА:

```
Непрочитанная часть          Состояние КА
10110101                     ЧЕТ
 0110101                     НЕЧЕТ
  110101                     НЕЧЕТ
   10101                     ЧЕТ
    0101                     НЕЧЕТ
     101                     НЕЧЕТ
      01                     ЧЕТ
       1                     ЧЕТ
 (пусто)                     НЕЧЕТ
```

Итак, КА заканчивает разбор цепочки в состоянии НЕЧЕТ, и, следовательно, цепочка 10110101 допускается.

Обычно КА задается таблицей. Например, для только что рассмотренного КА эта таблица будет выглядеть так:

	0	1	
ЧЕТ	ЧЕТ	НЕЧЕТ	0
НЕЧЕТ	НЕЧЕТ	ЧЕТ	1

Здесь:

- столбцы помечены входными символами (0 и 1);
- строки слева помечены символами состояний;
- первый элемент таблицы – начальное состояние;
- таблица состоит из символов новых состояний, которые соответствуют текущему состоянию (слева) и входному символу (сверху);
- строки справа помечены символами заключительных состояний (1 – допускающее, 0 – отвергающее).

Может показаться, что для решения такой простой задачи КА совершенно не нужен. Это верно, и в данном случае мы прекрасно могли обойтись и без него. Но на самом деле КА чрезвычайно полезны. Например, лексические анализаторы (сканеры) трансляторов обычно строятся именно как КА. Более того, существуют и широко исполь-

зуются утилиты, которые по описанию грамматики языка (о которых чуть ниже) могут автоматически создавать сканеры и даже синтаксические анализаторы. Известные многим и весьма популярные регулярные выражения (regular expressions) – это тоже КА.

Но КА, увы, не могут справиться со всеми проблемами трансляции. Например, КА не сможет разобрать цепочку вида 0^m1^m для произвольного m. Причина этого вполне очевидна – у КА нет способа убедиться в том, что число единиц совпадает с числом нулей. Все, что может КА, – это менять свое состояние в зависимости от текущего символа и текущего состояния. Можно сказать, что КА «не помнит» прошлого. Для произвольных значений m любой КА бессилен. Ему чего-то не хватает, но чего?

Собственно, только что ответ уже был дан: КА «не помнит» прошлого, т. к. у него нет памяти для хранения необходимой информации. Получается, если мы добавим КА памяти, то, может быть, он станет более универсальным? Это правильная гипотеза. Давайте добавим к КА память, и пусть это будет память, соответствующая дисциплине LIFO, т. е. стек. Но для начала вспомним, что в свое время мы вводили несколько основных операций над стеком (а именно push и pop). Но это еще не все. Мы собираемся объединить КА со стеком, и нам понадобится еще несколько операций. Сначала мы их просто опишем, а позже посмотрим в действии.

Для манипулирования содержимым вершины стека мы введем операцию замены replace (символы). Эта операция выталкивает элемент на вершине стека и проталкивает в стек указанные символы. Например, replace (abc) эквивалентно последовательности операций pop, push (a), push (b), push (c). Таким образом, операция replace – просто удобное сокращение последовательности операций pop и push.

Нам понадобятся еще две операции, но на этот раз над входной цепочкой. Операция shift (сдвинуть) означает продвижение к следующему символу цепочки; операция hold (держать) означает, что нужно продолжить обработку текущего символа цепочки (т. е. подождать).

В предыдущем примере конец входной цепочки определялся не самим КА, а программой, его реализующей. Но ничто не мешает ввести проверку на конец цепочки в сам КА. Множество символов, которые КА способен распознавать, называется его *алфавитом*.

Вот теперь мы можем составить КА, который будет в состоянии разобрать цепочку 0^m1^m для произвольного m. Этот КА будет использовать стек как память и управлять этой памятью. Напомним, что символ $\vdash$ обозначает дно стека. Символ для обозначения конца

входной строки не имеет печатного представления, но мы можем его заменить, например, символом ■. Конечный автомат, дополненный стековой памятью, мы будем называть стековым автоматом (или, сокращенно, С-автоматом). Этому С-автомату соответствует следующая *управляющая таблица*:

	0	1	■
X	replace(ZX) shift	pop hold	NO
Z	NO	pop shift	NO
⊦	NO	NO	YES

Здесь понадобятся некоторые пояснения. Управляющая таблица для С-автомата немного отличается от таблицы для КА. Слева указаны символы стека (в предыдущем примере здесь были указаны символы состояний); вверху – символы входной цепочки. Допускающие (YES) и отвергающие (NO) состояния являются частью таблицы. В некоторых ячейках таблицы необходимо выполнить операции (в указанной последовательности). Чтобы это стало более понятным, давайте проверим работу С-автомата на конкретной цепочке, например 000111:

```
Непрочитанная часть        Стек
000111■                    ⊦ X
 00111■                    ⊦ Z X
  0111■                    ⊦ Z Z X
   111■                    ⊦ Z Z Z X
   111■                    ⊦ Z Z Z
    11■                    ⊦ Z Z
     1■                    ⊦ Z
      ■                    ⊦
    (цепочка 000111 допускается)
```

Перед началом работы в стеке должен находиться символ X. Он выполняет роль маркера того, что С-автомат находится в процессе считывания нулей: всякий раз, когда из входной строки считывается 0, этот X заменяется на пару символов Z и X. После того как нули в цепочке заканчиваются, маркер больше не нужен (дальше нулей не должно быть). Ну а затем все происходит так, как мы уже не раз видели раньше.

Еще один пример С-автомата. Необходимо преобразовать произвольную цепочку нулей и единиц в цепочку вида 1^m0^n, т. е. цепочка 01101 должна быть преобразована в 11100. Вот управляющая таблица такого С-автомата:

	0	1	■
0	push(0) shift	print shift	print hold
├	push(0) shift	NO	YES

```
Непрочитанная часть        Стек
01101■                     ├
 1101■                     ├ 0
       print (1)
  101■                     ├ 0
       print (1)
   01■                     ├ 0
    1■                     ├ 0 0
       print (1)
     ■ print (0)           ├ 0
     ■ print (0)           ├
```

Здесь операция выталкивания из стека pop заменена операцией print, которая печатает содержимое вершины стека, а затем работает как операция pop.

Все только что рассмотренные операции могут быть решены проще, без использования КА и С-автоматов. Для чего нам тратить на них время и силы? Тут мы вынуждены извиниться: наша задача состояла не в том, чтобы дать основы трансляции языков программирования (это совсем другая тема) и применяемых здесь методов, а в том, чтобы в памяти читателя отложилась мысль, что такие методы существуют. Эти методы замечательно универсальны и пригодны для решения широкого класса задач.

Теперь поговорим о грамматиках и о том, как задаются (описываются) языки программирования.

Чтобы обеспечить точность перевода, языки программирования стремятся определить как можно точнее. Абсолютная точность в большинстве случаев не достижима, но это то, к чему следует стремиться.

Как можно определить язык программирования? Конечно, предложив его описание. Но тут возникает проблема – такое определение надо делать на уже имеющемся языке, т. е. на естественном! А он, как мы знаем, не однозначен.

Для того чтобы обеспечить строгость определений, мы можем прибегнуть к помощи математики, в частности способу передавать понятия в символьной форме. Давайте посмотрим, как это делается.

Начнем с примера: приведем формальное определение идентификатора (имен для подпрограмм, классов, переменных).

В любом учебнике по программированию мы прочтем, что «идентификатор – это последовательность букв и цифр, начинающаяся с буквы». То есть последовательность abc – это идентификатор, а #abc или a#bc – нет. А как будет выглядеть формальное определение идентификатора? Например, так:

```
<ID>     ::= <LETTER> {<LETTER>|<NUMBER>}
<LETTER> ::= A|B|C|...|Z|a|b|c|...|z
<NUMBER> ::= 0|1|2|...|9
```

Обратите внимание, что определяемое, т. е. ID, заключено в угловые скобки. Это так называемый *нетерминал*. Он, в свою очередь, определяется в терминах других нетерминалов (LETTER и NUMBER) одним из двух способов: либо это буква (LETTER), либо это буква, за которой следует произвольная последовательность букв или цифр (NUMBER); символ «|» читается как «или», а фигурные скобки означают, что записанное в них необязательно и может быть опущено.

Нетерминал LETTER – это или A, или B и т. д. до буквы z включительно. Нетерминал NUMBER – это или 0, или 1 и т. д. до цифры 9 включительно. Все буквы (A, B, C, ...) и цифры (от 0 до 9) – *терминалы*. Терминалы записываются как есть, т. е. буквально.

Нетерминалы – это категории, которые описывают целые классы. Каждый нетерминал рано или поздно должен быть заменен терминалами.

Ценность такого рода определений – не столько в краткости (что само по себе хорошо), сколько в том, что появляется возможность применять к анализу языков программирования строгие математические методы и даже строить на их основе программы-трансляторы.

Использованная выше форма записи носит название BNF, или форма Бэкуса–Наура (Backus и Naur), по фамилиям впервые предложивших эту форму специалистов.

Заметьте, что на некоторые вопросы это определение идентификатора ответов не дает. Например, различаются ли идентификато-

ры abc и ABC? Каково максимальное количество символов, входящих в состав идентификатора? При разработке трансляторов необходимо учитывать все факторы, даже те, для которых формальных определений нет.

Для полного описания языка не хватает только *правил* (в теории трансляции языков программирования обычно говорят о *продукциях*). Продукция – это указание на то, как из одних цепочек символов можно получать другие цепочки символов (вскоре мы рассмотрим пример, и станет понятнее, что такое продукция).

А теперь, полноты ради, мы приведем полное определение того, как задается *грамматика* языка. Грамматика задается:

- конечным множеством нетерминалов;
- конечным множеством терминалов;
- множества нетерминалов и терминалов не пересекаются;
- конечным множеством продукций вида <A> → α, где <A> – нетерминал и α – цепочка терминалов и нетерминалов (может быть пустая); <A> – это левая часть правила, а α – правая часть продукции;
- один из нетерминалов выделен как начальный.

Это определение задает т. н. *контекстно-свободную грамматику* (КС-грамматику); что такое КС-грамматика и какие еще есть грамматики, мы здесь обсуждать не будем. Достаточно того, что подавляющее большинство языков программирования задается именно такими грамматиками.

Замечательно здесь то, что аккуратно определенные грамматики допускают автоматический разбор (трансляцию) языков, в котором, разумеется, опять и опять появляется наш добрый и надежный друг – стек.

На этом мы завершаем последний раздел первой части книги. Мы понимаем, что очень многие вопросы остались без ответа. Как, например, связаны между собой грамматики и стеки? Как строятся соответствующие С-автоматы? Как происходит трансляция исходного кода? Как проверяется соответствие грамматики конкретным конструкциям в исходном коде? Как обрабатываются переменные? Как выделяется память? Какие при этом используются алгоритмы? Как обрабатываются ошибки? Как генерируется объектный код? И так далее...

Ответы на эти вопросы, к сожалению, не просты и выходят за рамки основной темы книги. Но они достойны внимания и того, чтобы потратить на их изучение силы и время.

За более подробной информацией о конечных автоматах, распознавателях, грамматиках и трансляции языков программирования следует обратиться к библиографии, к которой мы и адресуем читателей.

Заключение

Ну что же, мы проделали немалый путь и решили много интересных задач. Все эти задачи были взяты из практики, что добавляет убедительности тому, о чем мы говорили о стеке во введении: это действительно красивый и мощный инструмент, которым должен владеть программист.

Стек можно рассматривать как память с особой, времени й характеристикой: стек не просто хранит данные – он хранит данные с привязкой по времени. Именно это делает стек столь мощным, но в то же время простым и удобным.

Часть II

От слов – к делу

Расширенная стековая машина

В первой части книги нами были представлены очень простая каноническая стековая машина и ее система команд. Тогда же мы выяснили, что в качестве практического средства программирования эта машина мало пригодна: ее система команд бедна, а структуры управления зачаточные. Теоретически ее возможностей вполне достаточно, но практически программирование для нее совершенно непродуктивно.

Здесь мы попробуем исправить недостатки канонической стековой машины и, взяв за основу ее систему команд, разработаем новый вариант стековой машины. Будем называть такую машину расширенной (extended). В каких направлениях будет идти это расширение?

Для начала мы заменим символьные обозначения некоторых операций (таких, например, как !, @, R>, + и проч.) более привычными мнемоническими, приблизив их к тем обозначениям, которые обычно встречаются в ассемблерах.

Группу операций манипулирования стеком данных мы оставим без изменений:

Операция	Стековая диаграмма
DUP	├ ... a → ├ ... a a
DROP	├ ... a b → ├ ... a
SWAP	├ ... a b → ├ ... b a
OVER	├ ... a b → ├ ... a b a

Хотя существуют и другие операции, имеющиеся четыре покрывают большинство случаев. Если необходимость в других операциях

все же возникнет (прежде всего это команды циклической перестановки верхних трех элементов стека данных или чтение произвольного элемента стека), то их нетрудно реализовать, но мы не станем здесь этого делать.

Из системы команд канонической стековой машины мы полностью заимствуем арифметические и логические операции (в скобках приведены прежние обозначения), а также добавим некоторые полезные и часто встречающиеся операции:

Операция	Стековая диаграмма
ADD (+)	├ ... a b → ├ ... a+b
SUB (-)	├ ... a b → ├ ... a-b
MUL	├ ... a b → ├ ... a*b
DIV	├ ... a b → ├ ... a mod b a/b
NEG	├ ... a → ├ ... -a
ABS	├ ... a → ├ ... |a|
AND	├ ... a b → ├ ... a&a
OR	├ ... a b → ├ ... a|b
XOR	├ ... a b → ├ ... a^b
SHL	├ ... a → ├ ... a*2
SHR	├ ... a → ├ ... a/2

В число новых арифметических операций входят умножение MUL, деление DIV, а также NEG и ABS (изменение знака и абсолютная величина числа на вершине стека соответственно). Операция деления оставляет на вершине стека частное, а под ним – остаток от деления. Вот небольшой пример использования команды DIV:

```
21          ├ 21
4           ├ 21 4
DIV         ├ 1 5
```

(в правой колонке указаны состояния стека после выполнения каждой команды программы; если исходные числа делятся нацело, то остаток, равный 0, также будет протолкнут в стек).

Кроме того, мы добавили операции арифметических сдвигов влево и вправо на один разряд (что, соответственно, равносильно умножению и делению значения на вершине стека на 2); в сочетании с логическими операциями они позволяют производить достаточно сложные манипуляции с данными.

Уже имеющиеся команды дают возможность легко эмулировать простые и часто встречающиеся задачи. Допустим, нам необходимо

обнулить (очистить) элемент на вершине стека; глубина стека не должна меняться. Это можно сделать, например, так:

```
DUP                 0
SUB      или        MUL
```

(первая операция делает копию числа на вершине стека, вторая – находит разность, которая, конечно же, будет искомым 0).

А вот как можно увеличить или уменьшить значение на вершине стека на 1:

```
1                   1
ADD      или        SUB
```

Пока наши изменения в системе команд носили довольно «косметический» характер и не внесли ничего принципиально нового к тому, что у нас было прежде. Но сейчас будет и нечто совсем новое. Это новое касается структур управления.

Напомним, что в системе команд канонической стековой машины единственной структурой управления была операция IF. Эта операция, мягко говоря, не слишком удобна и плохо «приспособлена» для реализации различных типов переходов. Одна-единственная проверка (0 или не 0) сильно ограничивает и вынуждает вводить в программу довольно неестественные конструкции. Сейчас мы это поправим, введя вместо одной операции IF четыре новые. Одна из них – операция безусловного перехода, остальные три – операции условных переходов. Все они представлены в следующей таблице:

Операция	Стековая диаграмма
BR	⊢ ... addr → ⊢ ...
BRN	⊢ ... flag addr → ⊢ ...
BRZ	⊢ ... flag addr → ⊢ ...
BRP	⊢ ... flag addr → ⊢ ...

Операция BR (branch) выполняет безусловный переход по адресу на вершине стека, т. е. эта операция устанавливает memory [PC] = addr. Сам адрес перехода из стека выталкивается. Никакие условия при этом, разумеется, не проверяются.

Остальные три операции – BRN, BRZ и BRP – работают аналогично BR, но с одним существенным отличием. Каждая из этих операций ожидает в стеке два значения: адрес передачи управления (addr) на вершине стека и условие перехода flag под ней. Операция проверяет условие перехода; если условие перехода выполняется (меньше 0,

равно 0 или больше 0 соответственно), то операция выполняет передачу управления. Если условие перехода не выполняется, то будет исполняться следующая за операцией перехода операция. Вне зависимости, выполнится условие перехода или нет, и условие перехода, и адрес перехода из стека выталкиваются. Итак, теперь мы можем выполнять в программе передачи управления более «естественным» способом, зависящим от значения флага.

Операции управления подпрограммами, безусловно, очень полезны (они позволяют формировать структуру программы, разделяя ее на небольшие относительно самостоятельные модули), и мы их оставляем в нашей системе команд без изменений:

Операция	Стековая диаграмма (DS)	Стековая диаграмма (RS)
CALL	├ ... addr → ├ ...	├ ... → ├ ... raddr
RET	без изменений	├ ... raddr → ├ ...

Как и следовало ожидать, операция CALL, подобно операции BR, ожидает на вершине стека адрес начала подпрограммы. После выполнения операции CALL этот адрес выталкивается из стека. Операция RET обеспечивает возврат из подпрограммы, присваивая PC значение адреса возврата, после чего адрес возврата из стека возвратов выталкивается.

Операции >R и R> весьма полезны (если, конечно, применять их аккуратно, о чем мы говорили при описании канонической стековой машины), поэтому мы их оставляем и в нашей системе команд, изменив только мнемонику:

Операция	Стековая диаграмма (DS)	Стековая диаграмма (RS)
DTR	├ ... a → ├ ...	├ ... → ├ ... a
RTD	├ ... → ├ ... a	├ ... a → ├ ...

(DTR и RTD означают, соответственно, Data To Return и Return To Data).

Теперь обратимся к операциям работы с памятью (напомним, что в системе команд канонической стековой машины это были операции ! и @). Их общей особенностью является наличие на вершине стека данных адреса, по которому нужно сохранить или прочитать данные. Этот адрес должен быть известен до того, как будут выполнены операции ! и @. Следовательно, должен быть механизм для определения нужных адресов памяти и их проталкивания в стек. Это может быть проблемой.

Конечно, адрес памяти можно задать непосредственно в программе как число и протолкнуть его в стек операцией LIT. Это работает, но программист должен знать этот адрес памяти на этапе составления программы. Если программа изменится (в ней появятся новые фрагменты или, наоборот, что-то будет удалено), то не исключено (а скорее всего, именно так и будет), что этот адрес изменится. Кроме того, программа начинает зависеть от этих адресов; потом их будет совсем не просто изменить. Одним словом, задача определения адреса памяти может оказаться очень нетривиальной. Мы не будем менять семантику операций ! и @, а задачу определения адресов памяти переложим с программиста на компьютер. Раз численные значения адресов так или иначе надо определять, то пусть этим «займется» компьютер (как именно определяются нужные адреса, станет ясно чуть позже). Новые операции работы с памятью представлены в следующей таблице:

Операция	Стековая диаграмма
SAVE	├ ... n addr → ├ ...
LOAD	├ ... addr → ├ ... n

Операция SAVE ожидает на вершине стека данных адрес памяти, а под ней – данные, которые необходимо сохранить. Операция LOAD ожидает на вершине стека адрес памяти, из которого нужно загрузить данные в стек.

Теперь адрес памяти должен быть определен во время трансляции программы, т. е. автоматически. Как именно это происходит, мы увидим в одном из следующих разделов, а также в исходном коде транслятора (см. приложение). Здесь мы несколько забегаем вперед, так что пока примем это на веру.

Нам осталось рассмотреть совсем немного операций. Часть из них представлена в следующей таблице:

Операция	Стековая диаграмма
NOP	без изменений
IN	├ ... → ├ ... n
OUTN	├ ... n → ├ ...
OUTC	├ ... n → ├ ...
HALT	без изменений

Операция NOP (no operation) означает «ничего не делать». Такая возможность, при кажущейся бессмысленности, часто бывает полезной. Операция IN (input) позволяет организовать простейший спо-

соб ввода данных (а именно – чисел) в стек из консоли во время исполнения программы. Операции OUTN и OUTC (OUT as Number/Character) распечатывают в консоли значение элемента на вершине стека данных, соответственно, как число или как символ. После выполнения операции OUTC перевод строки не производится. Это позволяет формировать строку из выводимых символов. Для того чтобы выполнить перевод строки, нужно указать специальный управляющий код (10) и вывести его как символ. Важно помнить, что выводимое значение, как обычно, удаляется из стека данных.

Операция HALT останавливает исполнение программы.

Мы исключили из системы команд операцию LIT. Это может показаться нелогичным: каким образом теперь можно будет проталкивать в стек заранее определенные значения? Очень просто: мы сделаем эти значения частью программы (выше, при обсуждении команды DIV, мы привели соответствующий пример). Таким образом, если в программе встретится число 10, то оно будет автоматически протолкнуто в стек данных. Правда, тут есть одна тонкость, связанная с отрицательными числами, о которой мы расскажем, когда будем описывать работу транслятора.

Нам осталось рассмотреть еще две операции. Их значимость неочевидна, но порой они могут оказаться незаменимыми:

Операция	Стековая диаграмма
LPC	├ ... → ├ ... PC
DEPTH	├ ... → ├ ... n

Операция LPC (от Load PC) проталкивает на вершину стека текущее значение программного счетчика. Операция DEPTH проталкивает на вершину стека данных его текущую глубину. Если стек данных пуст, то операция DEPTH оставит в стеке значение 0 (после чего, разумеется, стек данных уже не будет пустым). Таким образом мы можем программным способом проверять стек данных на исчерпание.

Такова система команд расширенной стековой машины. Она несколько больше рассмотренной ранее системы команд канонической стековой машины. Предыдущая версия системы команд была более компактной, но неудобной; многие утилитарные, но полезные операции в ней отсутствовали. Новая и дополненная система команд, безусловно, гораздо практичнее.

Несколько слов – о режимах адресации расширенной стековой машины. Мы старались сделать так, чтобы и транслятор, и интерпрета-

тор получились максимально простыми, поэтому был предусмотрен лишь непосредственный режим адресации (при котором доступ к содержимому ячейки памяти осуществляется явно, т. е. по номеру или адресу этой ячейки, указанному в программе).

Ценность языка программирования в конечном счете определяется тем, насколько этот язык пригоден для решения задач. Разумеется, во всем нужна мера; не следует пытаться определить язык так, чтобы предусмотреть в нем все, что теоретически может понадобиться. Это попросту невозможно, и такая попытка быстро превратит любой язык в сложную, громоздкую и неудобную конструкцию. В языке, чтобы он мог считаться если не превосходным, то хотя бы полезным, необходимо иметь достаточно полный (но не избыточный) набор конструкций, а также возможности, позволяющие расширить этот язык в нужном направлении.

На этом мы завершим описание системы команд расширенной стековой машины. Теперь настало время изучить устройство транслятора, но для начала, на примере простой программы, посмотрим сеанс работы стековой машины.

Первая программа

Ниже представлен исходный код нашей первой программы:

```
#################################################################
# Факториал n! (нерекурсивный способ)                           #
#################################################################
           1
           IN                   ; Ввести число n (= счетчик)
           ABS

# Основная часть программы
:loop      DUP
           exit                 ; Счетчик повторений == 0?
           BRZ                  ; Да, остановить
           SWAP                 ; Копия
           OVER                 ; счетчика
           MUL                  ; fact * count
           SWAP                 ; Восстановить счетчик
           1                    ; Уменьшить счетчик на 1
           SUB
           loop                 ; Повторить
           BR
```

```
# Вывод результата
:exit     DROP                  ; Лишний счетчик повторений
          OUTN                  ; Вывести результат
          HALT
#################### Конец программы ########################
```

Эта программа предлагает ввести в консоли число и находит его факториал.

Символы # и ; служат признаками комментариев: все, что следует за ними до конца строки (включая их самих), игнорируется.

Мы прокомментировали каждую строку программы, но, конечно, на практике не стоит быть настолько многословными; достаточно комментировать только то, что действительно нужно.

Если запустить стековую машину на исполнение (как именно это делается, мы расскажем в приложении) и указать имя исходного файла, содержащего эту программу, то будет выведен следующий протокол (то, что вводится пользователем, выделено подчеркиваниями):

```
++++++++++++++++++++++++++++++++
+ Assembler for ToyStackMachine +
++++++++++++++++++++++++++++++++
Source file name: fact
Parsing (pass 1): ok
Parsing (pass 2): ok

OBJECT CODE
----------------------
000000              1
000001            -26
000002            -12
000003             -1
000004             14
000005            -20
000006             -3
000007             -4
000008             -9
000009             -3
000010              1
000011             -8
000012              3
000013            -18
000014             -2
000015            -27
000016            -40
```

```
SYMBOL TABLE
----------------------
EXIT            000014
LOOP            000003

Run...
Number: 10
3628800
Program completed

CONTENTS OF
[       EXIT] = -2
[       LOOP] = -1
   dsp = -1
   rsp = -1
    pc = 17
cycles = 119
```

(т. о. факториал 10 равен 3 628 800).

Протокол состоит из нескольких разделов. Опишем их по порядку.

После ввода имени исходного файла (fact) начинается трансляция программы. Если трансляция завешается успешно (т. е. в исходном коде нет ошибок), то генерируется и выводится в консоль объектный код (object code). Объектный код распечатывается в виде двух столбцов: в левом – адреса основной памяти, в правом – объектный код. Из этой распечатки видно, что объектный код совсем небольшой: всего 17 ячеек памяти (с 0 по 16).

В исходном коде имеются две метки, каждая из которых начинается с символа двоеточия (:). Метки позволяют сопоставить имена (переменные) адресам памяти. Такое сопоставление запоминается в небольшой, но исключительно важной таблице – таблице символов, которая тоже выводится в консоль.

Наверное, было бы правильнее назвать эту таблицу просто таблицей меток, но по историческим причинам используется термин таблица символов.

В таблице символов, как и в объектном коде, два столбца. В левом – имена меток, в правом – соответствующие им адреса памяти. Таким образом, из таблицы символов мы, к примеру, видим, что метке exit соответствует адрес 14.

Значение меток невозможно переоценить; можно смело сказать, что это самый полезный элемент для программирования на ассемблере. Метки освобождают программиста от утомительной и непродук-

тивной работы отслеживания адресов и учета используемой памяти. Вместо того чтобы самому следить за тем, какие в программе используются адреса, мы просто снабжаем метками нужные части программы, а все дальнейшее – забота транслятора.

Затем программа начинает исполняться. В консоль выводится приглашение для ввода числа, факториал которого нужно вычислить.

По завершении работы в консоль выводится разнообразная полезная информация: содержимое стеков данных и возвратов (если только они не пусты), содержимое адресов памяти, для которых имеются метки, значения основных регистров и, наконец, общее число циклов выборки и исполнения команд.

Вся информация, выводимая в протокол, очень полезна (особенно если опыт программирования для стековой машины не велик), и хотя ее можно отключить, внеся совсем небольшое изменение в исходный код стековой машины, мы настоятельно не рекомендуем этого делать.

Описывать, как именно эта программа вычисляет факториал, мы не станем. На самом деле программа очень проста (в ней фактически реализуется аналог цикла со счетчиком, причем счетчиком выступает число, факториал которого ищется). Самый лучший способ понять, что здесь происходит, – попробовать разобраться самостоятельно. Для этого нужно «пройтись» по каждой команде и на каждом шаге выписать содержимое стека данных до и после операции (в этом варианте подпрограммы и, следовательно, стек возвратов не использовались).

Чуть позже мы покажем еще несколько примеров программ для нашей стековой машины. Все эти примеры намеренно просты; их цель – не столько научить программированию стековой машины (программированию вообще нельзя научить, а можно лишь помочь), сколько предоставить ряд образцов, отталкиваясь от которых, можно попробовать решить другие, более интересные, но и более сложные задачи.

Как работает транслятор

Сейчас мы кратко опишем работу транслятора расширенной стековой машины. Это описание призвано помочь тем читателям, кто захочет внести в транслятор и интерпретатор стековой машины изменения (например, новые операции, режимы адресации, средства отладки, трассировки, возможность сборки программы из раздельно транслируемых модулей и проч.).

Исходный код реализации расширенной стековой машины включает в себя всего 5 файлов и занимает в распечатанном виде немногим

больше 700 строк на языке программирования Java. Это совсем немного. Правда, такая краткость была вынужденной: желая сохранить исходный код транслятора понятным и легко обозримым, мы должны были пожертвовать рядом полезных возможностей, которые обычно встречаются в ассемблерах. Большую их часть добавить совсем не трудно, но осознанно мы не стали этого делать (по причинам, изложенным выше) и потому предоставляем читателю право изменять исходные коды по своему усмотрению.

Класс ToyStackMachine.java

В этом классе находится точка входа, с которой начинается исполнение программы, т. е. метод main. Сначала запрашивается имя файла с исходным кодом программы (см. метод getFileName). Затем содержимое этого файла (см. метод loadSource) считывается в списочный массив (объявленный как ArrayList <String>). Осуществляются некоторые типичные проверки при работе с файлами; если ошибок не было, то загруженный исходный код передается ассемблеру для трансляции.

Класс OpcodeTable.java

Это очень простой класс, в котором перечислены все операции, входящие в систему команд стековой машины. Единственная тонкость связана с кодами операций: все они – целые отрицательные числа. Вспомним, что несколько раньше мы говорили о том, что с отрицательными числами в исходных кодах программ для стековой машины надо быть внимательными. Сейчас мы объясним, почему так.

Когда интерпретатору в объектном коде «встречаются» неотрицательные числа (т. е. 0 и положительные), то эти числа сразу же проталкиваются в стек данных (именно так и происходило выше в примере с операцией DIV). Вот почему в системе команд нашей машины нет операции LIT, которая присутствовала в канонической стековой машине.

Но когда интерпретатору в объектном коде «встречаются» отрицательные числа, он пытается сопоставить их с кодами операций. Если это ему удается, то операция выполняется.

Пусть, например, в объектном коде встретилось число –8. Обратившись к кодам операций, мы видим, что этому числу соответствует операция SUB (вычитание). Теперь интерпретатор пытается выполнить эту операцию (конечно, при условии, что в стеке данных имеются хотя бы два числа).

Но если интерпретатору в объектном коде встретится число –57, то теперь все гораздо хуже: числу –57 не соответствует ни одной операции. Что теперь делать интерпретатору? Здесь есть три варианта: либо игнорировать это число, либо протолкнуть его в стек данных, либо завершить работу программы с ошибкой.

Мы выбрали последний вариант, т. е. завершение работы программы. Почему? Игнорировать такое число было бы нелогичным: раз число есть в исходном коде, то оно было для чего-то написано. Для чего – не понятно, и лучшее, что тут можно сделать, – предложить программисту еще раз поработать с исходным кодом. Проталкивать в стек? Тоже вариант так себе: какие-то числа на самом деле соответствуют операциям и вызывают действия, а какие-то почему-то нет и должны попасть в стек данных. Странная привилегия.

Поэтому если в стек данных нужно занести отрицательное число, то лучше воспользоваться операцией NEG, а в исходном коде использовать только положительные числа.

В связи с этим мы расскажем об одном программистском трюке, который, однако, рекомендуем использовать лишь в исключительных случаях, т. е. тогда, когда никаких других вариантов решения задачи уже нет. Рассмотрим следующую программу:

```
40              ├ ... 40
NEG             ├ ... -40
1000            ├ ... -40 1000
SAVE            ├ ...
1000            ├ ... 1000
BR
```

Что произойдет после исполнения операции безусловного перехода BR на адрес 1000? Давайте подумаем...

Вначале все просто: в стек данных проталкивается положительное число (40), и изменяется его знак. Затем это число (т. е. –40) сохраняется в памяти по адресу 1000. Пока ничего «криминального». Но вот управление получает операция BR. С вершины стека данных снимается адрес перехода, после чего управление передается по адресу 1000. А там, как мы помним, находится число –40. Поскольку число отрицательное, то интерпретатор считает его кодом операции и пытается определить, какой именно операции это число соответствует. Это число соответствует операции HALT, и программа завершает свое исполнение. Может, так и было задумано, но больше походит на ошибку.

И еще один весьма распространенный способ, но на этот раз связанный с изменением сгенерированных объектных кодов.

Объектные коды после трансляции загружаются в основную память и занимают в ней ячейки с адресами от 0 до некоторой величины (скажем, в примере нашей первой программы объектный код был загружен в ячейки с адресами от 0 до 16). Ничто не мешает изменить содержимое любой из этих ячеек самой же программе способом, подобным тому, что был приведен выше. Такая программа называется самомодифицируемой. В обычных условиях необходимость в подобных модификациях практически отсутствует, поэтому следует быть внимательным: можно потратить массу времени и сил, пытаясь понять, почему программа не работает или работает неправильно. Вывод здесь простой: перед тем как прибегать к таким методам, лучше еще подумать и поискать другое решение. В последующем, при изменении программы, это окупит и усилия, и время.

Если внимательно присмотреться к кодам операций, то видно, что между операциями –31 (NOP) и –40 (HALT) оставлен промежуток. Этот «зазор» не случаен: в него можно добавить новые операции. Для этого, правда, придется внести еще и изменения в исходный файл интерпретатора, но об этом мы поговорим несколько позже.

Класс SymbolTable.java

Этот небольшой класс играет ключевую роль. Здесь хранится информация о метках и назначенных каждой метке адресах памяти (обычно говорят не об адресах, а о *значениях* меток). В классе реализовано несколько методов. Метод lookupSymbol определяет, имеется ли метка с заданным именем в таблице символов. Метод putSymbol добавляет информацию о метке в таблицу. Метод getValue возвращает значение метки по ее имени. Наконец, простой метод printSymtable распечатывает таблицу символов (но только в том случае, когда таблица символов непуста, т. е. содержит хотя бы одну метку); пример распечатки мы видели ранее, в разделе, где обсуждалась первая программа.

Нам осталось рассмотреть последние два класса. Это – самые большие классы, но они достаточно просты.

Класс Assembler.java

В этом классе осуществляется трансляция исходного кода в объектный код. Трансляция построена по классической двухпроходной схеме, при которой исходный код читается дважды.

Все символы исходного кода программы преобразуются транслятором в верхний регистр. Поэтому транслятор считает операции HALT и halt идентичными; то же самое относится и к меткам: метки LABEL и label не отличаются друг от друга.

На первом проходе обрабатываются метки. Почему для обработки меток необходим отдельный проход? Причина одна, но очень важная.

Дело в том, что в программе может встретиться передача управления по адресам, которые расположены в старших адресах памяти. Например, в программе вычисления факториала, рассмотренной ранее, имеется следующий фрагмент:

```
            ...
            exit                ; Счетчик повторений == 0?
            BRZ                 ; Да, остановить
            ...
:exit       DROP                ; Лишний счетчик повторений
```

Здесь происходит передача управления к метке exit, которая расположена где-то ниже. Вместо имени метки exit в объектный код нужно подставить значение метки, т. е. ее адрес. Но этот адрес необходимо сначала определить, и именно для этого нужен отдельный (первый) проход по исходному коду.

При втором проходе генерируется объектный код. Транслятор взаимодействует с двумя (справочными) таблицами: с таблицей операций и с таблицей символов (меток). Сама генерация объектного кода очень проста и сводится к подстановке вместо имен меток их значений, а вместо мнемоник операций – их числовых кодов.

В сущности, важно разобраться с одним методом – getToken. Этот метод извлекает из исходного кода программы строку и выделяет из нее лексемы, т. е. отдельные составляющие. Всего предусмотрены пять типов лексем, и все они описаны в виде констант в начале файла.

Лучший способ разобраться с транслятором – внимательное изучение его кода и анализ результатов, которые транслятор выводит в консоль.

В библиографии мы указали некоторые книги, посвященные трансляции языков, подобных языку расширенной стековой машины. Мы рекомендуем ими воспользоваться.

Транслятор может быть достаточно просто расширен в желаемом направлении. В него можно ввести, например, директивы (т. е. указания) резервирования памяти, различные режимы адресации и прочие полезные расширения.

Класс StackMachine.java

Этот класс представляет собой интерпретатор объектного кода, который был сгенерирован транслятором. Это простой, хотя и достаточно объемный класс. Обработка объектных кодов организована в виде большого switch-оператора; каждому объектному коду (т. е. его оператору) соответствует своя case-ветка.

Здесь мы обсудим лишь то, как организована память и как осуществляется управление стеками данных и возвратов. Посмотрим на следующий фрагмент:

```
static final int MEMORYSIZE = 65536;     // Объем основной памяти
private int memory [] = new int [MEMORYSIZE],
            dstack [] = new int [256],  // Стек данных
            rstack [] = new int [256];  // Стек возвратов

private int loaded = 0,  // Длина объектного кода
            dsp    = -1, // Указатель вершины стека данных
            rsp    = -1, // Указатель вершины стека возвратов
            pc     = 0,  // Программный счетчик
            cycles = 0;  // Число циклов исполнения
```

Здесь прежде всего задаются объемы основной памяти и стеков. Мы выбрали для них совсем небольшие числа, но их вполне достаточно для работы нетривиальных программ среднего размера. Разумеется, эти величины можно изменить как в большую, так и в меньшую стороны.

Стеки данных и возвратов реализованы как массивы фиксированной длины. В «настоящих» трансляторах этого стараются избегать, а стеки размещают в динамической памяти. Тогда для стеков будет выделено столько места, сколько нужно.

Несколько слов о доступе к вершине стека (на примере стека данных). Для каждого стека предусмотрен отдельный указатель, т. е. переменная, которая указывает на вершину стека. В пустом стеке указатель стека равен –1, т. е. фактически никуда не указывает. При проталкивании нового элемента в стек данных указатель должен быть скорректирован следующим образом: dstack [++dsp]. При выталкивании элемента из стека указатель необходимо изменить: dsp--. Доступ к элементу на вершине стека осуществляется посредством dstack [dsp].

Многочисленные примеры работы с указателями стеков можно найти в case-ветках оператора switch.

Расширение системы команд

Ранее мы говорили о том, что между операциями NOP и HALT оставлен «зазор», который можно использовать для добавления в систему команд новых операций. Вот как это делается.

Допустим, нам нужна операция увеличения значения на вершине стека данных на 1 (инкремент). Это действительно полезная операция, и она весьма удобна для программирования циклов.

Сначала эту операцию нужно «зарегистрировать» в системе команд стековой машины (файл OpcodeTable.java):

```
...
optable.put («NOP»,   -31);          // Нет операции
optable.put («INC»,   -32);          // Инкремент
// Завершение работы
optable.put («HALT»,  -40);          // Остановить
...
```

Мы просто добавляем новую команду, указывая ее мнемонику и уникальный код операции.

Теперь нужно немного поправить интерпретатор (StackMachine.java):

```
...
// Нет операции
case -31:
    break;
// Инкремент
case -32:
    dstack [dsp] = dstack [dsp] + 1;
    break;
// Остановить
case -40:
    System.out.printf ("Program completed%n");
    break loop;
...
```

Теперь осталось перекомпилировать исходный код на Java и проверить работу транслятора и интерпретатора на простом примере:

```
10
INC
OUTN
HALT
```

Если все было сделано аккуратно, то в консоль будет выведено число 11. Вот и все – новая операция добавлена, и ее можно исполь-

зовать в своих программах точно так же, как и те операции, что были изначально.

Примеры программ

Мы не ставим перед собой цель научить программированию на расширенной стековой машине. Да это, честно говоря, и невозможно: научиться программированию можно лишь одним способом – писать программы. Но, желая облегчить эту непростую работу, мы приведем несколько примеров программ. Мы не стремились к тому, чтобы составить эти примеры самым оптимальным образом. В отсутствие достаточного опыта и навыков такие оптимизированные программы способны только запутать начинающих.

Суммирование последовательности чисел

```
# Чтение и сложение чисел из консоли, пока не будет введен 0

; Основная программа (цикл ввода чисел)
:loop  IN                         ; Ввод числа
       DUP                        ; Копия
       exit                       ; Адрес перехода
       BRZ                        ; если 0
       summa                      ; Вызвать
       CALL                       ; подпрограмму суммирования
       loop                       ; На начало
       BR                         ; ввода чисел

; Завершение работы программы
:exit  total                      ; Загрузить
       LOAD                       ; общую сумму
       OUTN                       ; Вывести
       HALT                       ; Завершить

; Подпрограмма накопления суммы введенных чисел
:summa total                      ; Загрузить
       LOAD                       ; текущую сумму
       ADD                        ; Прибавить очередное число
       total                      ; Сохранить
       SAVE                       ; текущую сумму
       RET                        ; Вернуться

; Здесь будет накапливаться сумма
:total 0

# Конец программы
```

Эта программа позволяет ввести в консоли произвольное количество чисел (как положительных, так и отрицательных) и найти их сумму. Условие завершения – ввод числа 0.

Сложение двух чисел (вариант 1)

Хотя в системе команд имеется операция ADD, мы покажем, как осуществляется сложение двух чисел (второе из которых положительно) другим способом; о нем мы упоминали в первой части книги в разделе о передаче параметров:

```
# Нерекурсивное сложение двух чисел путем прибавления единиц
# в том количестве, которое содержится во втором слагаемом

        first
        LOAD            ; загрузить первое слагаемое
:loop   second
        LOAD            ; загрузить второе слагаемое
        DUP             ; копия второго слагаемого (для проверки на 0)
        stop
        BRZ             ; перейти, если второе слагаемое равно 0
        sum
        CALL            ; перейти к подпрограмме суммирования
        loop
        BR              ; на начало цикла
:stop   DROP            ; лишнее слагаемое
        OUTN            ; вывод
        HALT

; подпрограмма суммирования
:sum    1
        SUB             ; уменьшить на 1 второе слагаемое
        second
        SAVE            ; сохранить его
        1
        ADD             ; увеличить на 1 первое слагаемое
        RET             ; возврат в основную программу

; слагаемые
:first  -10
:second 3000000

# Конец программы
```

Здесь к первому слагаемому (число –10) прибавляются единицы в количестве, которое содержится во втором слагаемом (т. е. 3 000 000 раз). Как мы видим, сумма находится довольно быстро (хотя, бесспор-

но, все равно неэффективно, по сравнению с операцией ADD). В этом примере показано использование подпрограммы (нерекурсивной).

Сложение двух чисел (вариант 2)

А вот это – рекурсивный вариант предыдущей программы:

```
# Рекурсивное сложение двух чисел путем прибавления единиц
# в том количестве, которое содержится во втором слагаемом

          first       ; Первое число -> в стек данных
          LOAD
          rec         ; Вызов рекурсивной подпрограммы
          CALL
          OUTN
          HALT

# Добавить к содержимому стека данных количество единиц,
# содержащихся во втором слагаемом (например, 3 = 1 1 1)
:rec      1           ; Протолкнуть 1 в стек данных
          ADD         ; Прибавить 1
          second      ; Уменьшить второе слагаемое на 1
          LOAD
          1
          SUB
          DUP
          second
          SAVE
          end
          BRZ         ; Если второе слагаемое равно 0, то выход
          rec         ; Иначе - рекурсивный вызов
          CALL
:end      RET

# Первое слагаемое
:first    -21
# Второе слагаемое
:second   100

# Конец программы
```

Здесь мы опять складываем два числа (второе из которых должно быть положительным). Обратите внимание на то, что в программе есть рекурсивный вызов подпрограммы:

```
          ...
          rec         ; Иначе - рекурсивный вызов
          CALL
          ...
```

С этой программой придется «повозиться». Пожалуй, единственный способ понять, как она работает, – тщательно проследить за состоянием стека данных и стека возвратов. Условие прекращения рекурсивных вызовов – нулевое значение второго слагаемого.

Здесь же можно посмотреть, что произойдет, если число рекурсивных вызовов превысит допустимую глубину стека возвратов. Эта глубина ограничивается емкостью стека возвратов. Измените второе слагаемое (например, на 257) и посмотрите, что произойдет.

Сложение последовательности чисел в памяти

Следующая программа находит сумму чисел, хранящихся в памяти:

```
# Сумма чисел в памяти
          begin     ; Адрес начала последовательности
          end       ; Адрес конца последовательности
          SWAP      ; Установить правильный порядок перед вычитанием
          SUB       ; Теперь в стеке значение (END - BEGIN) = 5
          count     ; Сохранить счетчик повторений
          SAVE
          0         ; 0 -> в стек данных
          summa
          CALL
          HALT

# Подпрограмма суммирования последовательности чисел
:summa    NOP
:loop     begin     ; Смещение к очередному числу от BEGIN
          count
          LOAD
          ADD       ; Адрес следующего числа в последовательности
          LOAD
          ADD       ; Прибавить очередное число последовательности
          count     ; Скорректировать счетчик
          LOAD
          1
          SUB
          DUP
          count
          SAVE
          done      ; Все числа просуммированы (счетчик < 0)?
          BRN
          loop      ; Следующее число
          BR
  :DONE   RET
```

```
# Тестовые данные
:begin    100
          -90
           84
         1027
            0
          900
           -5
:end       61

:count      0
# Конец программы
```

Это довольно большая, но не сложная программа. Важно понять, как мы определяем количество чисел, подлежащих суммированию: начало последовательности чисел имеет метку begin, а конец – метку end. Результат суммирования не выводится в консоль, а остается в стеке данных.

Факториал

Ранее мы приводили программу вычисления факториала. Теперь посмотрим на рекурсивный вариант этой же программы:

```
# Факториал n! (рекурсивный способ)

            1
            IN                  ; Ввести число
            ABS
            count               ; Сохранить копию
            SAVE
            fact
            CALL
            DROP                ; Лишний счетчик повторений
            OUTN                ; Вывести результат
            HALT

 :fact      count
            LOAD
            DUP
            done
            BRZ
            MUL
            count
            LOAD
            1
```

```
        SUB
        count
        SAVE
        fact
        CALL
:done   RET

:count  0                       ; Счетчик повторений
# Конец программы
```

Условием завершения рекурсии является равенство нулю значения count.

Эта программа стоит того, чтобы ее внимательно изучить. В ней есть что улучшать, но перед тем, как попытаться сделать это, нужно основательно разобраться во всех деталях и, главное, в том, как завершаются рекурсивные вызовы подпрограммы.

Вывод текста

Ниже приводится исходный код программы, которая выводит текстовое сообщение в консоль:

```
# Вывод текстового сообщения

        begin       ; Адрес начала последовательности символов
        end         ; Адрес конца последовательности символов
        SWAP        ; Установить правильный порядок перед вычитанием
        SUB         ; Теперь в стеке значение end - begin
        count       ; Сохранить счетчик повторений
        SAVE
        print
        CALL
        HALT

# Подпрограмма вывода символов
:print  NOP
:loop   begin       ; Смещение к очередному числу от begin
        count
        LOAD
        ADD         ; Адрес следующего числа в последовательности
        LOAD
        OUTC        ; Напечатать очередной символ
        count       ; Скорректировать счетчик
        LOAD
        1
        SUB
        DUP
```

```
         count
         SAVE
         done          ; Все символы выведены (счетчик < 0)?
         BRN
         loop          ; Следующее число
         BR
:done    RET

# ASCII-коды символов сообщения Hello,World!
:begin   10            ; Перевод строки
         72            ; Код символа H
         101           ; Код символа e
         108           ; Код символа l
         108           ; и т. д.
         111
         44
         87
         111
         114
         108
         100
:end     33

:count   0
# Конец программы
```

В этой программе есть одна недоработка: сообщение «Hello,World!» выводится в перевернутом виде, т. е. как «!dlroW,olleH». Эту недоработку мы предлагаем устранить самостоятельно.

Программа-шутка

Напоследок мы приведем совсем небольшую программу. Попробуйте, не запуская ее на исполнение, определить, какую задачу она решает. Потом запустите и посмотрите результат:

```
# Программа-шутка
        IN          ; Ввести числа
        IN
        OVER        ; Сделать их копии в стеке данных
        OVER
        ADD         ; a+b
        DTR         ; Сохранить в стеке возвратов
        SUB         ; a-b
        ABS         ; |a-b|
        RTD         ; Восстановить из стека возвратов
        ADD         ; a+b-|a-b|
```

```
        SHR        ; Сдвиг вправо (/2)
        OUTN
        HALT
# Конец программы
```

Программы такого рода по-своему изящны, но их следует использовать с осторожностью: глядя на них, очень трудно понять, что они делают. Если же искушение слишком велико, то такие программы следует тщательно комментировать. Иначе по прошествии совсем небольшого времени мы сами не разберемся, что здесь происходит.

В программе демонстрируются операции передачи данных из стека данных в стек возвратов и обратно. Кроме того, здесь используется операция сдвига вправо.

Указания по программированию

Главная сложность программирования для стекового компьютера заключается в том, что программист должен очень внимательно следить за состоянием стеков (в особенности стека данных). Например, нужно помнить, что всякий раз, когда происходит передача управления, в стек данных необходимо протолкнуть адрес перехода. Вне зависимости от того, было передано управление или нет, этот адрес из стека данных будет вытолкнут.

По мере накопления опыта будут вырабатываться и необходимые навыки. Это подобно тому, как ребенок учится ходить. Сначала он ходит, поддерживаемый родителями, потом сам начинает цепляться – за стены и мебель и, наконец, отрывается от них и идет свободно. Точно так и в программировании; поначалу все идет трудно и порой даже мучительно, но постепенно мы обучаемся. Такой навык можно приобрести только одним способом – решая задачи.

К вершинам мастерства

Вот примерный список задач, на которых можно попробовать свои силы (уровень сложности задач постепенно возрастает к концу списка):

- наибольший общий делитель двух положительных чисел;
- квадраты чисел от m до n;
- добавить в систему команд операцию циклической перестановки трех верхних элементов стека ROT (⊦ ... a b c → ⊦ ... b c a);
- задано число; определить – простое это число или составное;
- поместить на вершину стека n-й элемент стека, считая с 0 (для проверки: если n = 0, то это эквивалентно операции DUP; если

n = 1, то это эквивалентно операции OVER). Перед извлечением n-го элемента стека необходимо удостовериться, что это возможно (т. е. проверить глубину стека);

- дана последовательность (массив) чисел; сформировать новую последовательность, в которой каждое четное число из исходной последовательности будет поделено на 2, а каждое нечетное – умножено на 7;
- найти все целые делители числа 60;
- возведение числа в целую положительную степень;
- скопировать данные из одной области памяти в другую;
- перепишите программу-шутку, используя более традиционный подход (программа должна работать для любых комбинаций чисел – и положительных, и отрицательных);
- дано количество часов, минут и секунд; перевести их в секунды (а затем обратно – в часы, минуты и секунды);
- дана последовательность n чисел, каждому числу сопоставлен индекс (от 0 до n – 1); умножить каждое число на его индекс;
- последовательность Фибоначчи для заданного целого положительного числа (нерекурсивный и рекурсивный варианты);
- правильность скобочной структуры (здесь надо придумать, как скобочная структура будет представлена в программе);
- пузырьковая сортировка последовательности чисел;
- удалить дубликаты в последовательности чисел, оставив только по одному экземпляру каждого числа;
- напишите программы, которые реализуют стек способами, описанными в приложении А.

Разумеется, список задач можно продолжать и продолжать. Мы убеждены, что читатель не только рано или поздно справится с любой из предложенных задач, но и сам найдет, придумает и реализует другие задачи, соответствующие его вкусу и его интересам.

Заключение

На этом заканчивается вторая часть книги о стеке, и настало время попрактиковаться в составлении программ для стекового компьютера. Работа предстоит не простая, но увлекательная.

Автор надеется, что ему удалось передать на страницах этой небольшой книги свое восхищение стеком – на первый взгляд, незатейливой и простой структурой данных, которая тем не менее способна на столь многое.

Библиография

Для того чтобы читателю легче было ориентироваться, ссылки в библиографии разделены на несколько категорий.

Главные

1. *Ахо А., Ульман Дж., Сети Р.* Компиляторы: принципы, технологии, инструменты. М.: Вильямс, 2001.
2. *Вирт Н.* Построение компиляторов. М.: ДМК Пресс, 2010.
3. *Кнут Д.* Искусство программирования. Т. 1: Основные алгоритмы. 3-е изд. М.: Вильямс, 2004.
4. *Пратт Т., Зелковиц М.* Языки программирования: разработка и реализация. 4-е изд. СПб.: Питер, 2002.

Стек и язык программирования Forth

1. *Brodie L.* Thinking Forth. URL: thinking-forth.sourceforge.net.
2. *Conklin E., Rather E.* Forth Progtammer's Handbook. Sixth edition. California (USA): FORTH Inc., 2000. URL: www.forth.com.
4. *Koopman Ph.* Stack Computers. The new Way. Mountain View Press (USA), 1989. URL: users.ece.cmu.edu/~koopman/stack_computers/.
5. *Loeliger R. G.* Threaded_interpretive_languages. URL: http://sinclairql.speccy.org/archivo/docs/books/Threaded_interpretive_languages.pdf.
6. *Броуди Л.* Начальный курс программирования на языке Forth. М.: Финансы и статистика, 1990.
7. *Тоффоли Т., Марголус Н.* Машины клеточных автоматов. М.: Мир, 1991.

Ассемблеры и вопросы трансляции

1. *Баррон Д.* Ассемблеры и загрузчики. М.: Мир, 1974.
2. *Бек Л.* Введение в системное программирование. М.: Мир, 1998.
3. *Хантер Р.* Проектирование и конструирование компиляторов. М.: Финансы и статистика, 1984.

Операционные системы

1. *Лав Р.* Разработка ядра Linux. 2-е изд. М.: Вильямс, 2006.
2. *Танебаум А., Вудхалл А.* Операционные системы: разработка и реализация. 3-изд. СПб.: Питер, 2007.

Прочее, но отнюдь не второстепенное

1. *Абельсон Х., Сассман Дж.* Структура и интерпретация компьютерных программ. М.: Добросвет; КДУ, 2006.
2. *Каррано Ф., Причард Дж.* Абстракция данных и решение задач на C++. 3-е изд. М.: Вильямс, 2003.
3. *Стюарт Т.* Теория вычислений для программистов. М.: ДМК Пресс, 2014.
4. *Хендерсон П.* Функциональное программирование: применение и реализация. М.: Мир, 1983.
5. *Хоггер К.* Введение в логическое программирование. М.: Мир, 1988.
6. *Эккель Б.* Философия Java. 4-е изд. СПб.: Питер, 2011.

Приложение A

Способы реализации стеков

Ниже мы приводим описание двух основных способов реализации стека: на основе массива и связанного списка.

Реализация стека на основе массива

Массив задается двумя характеристиками: максимальным размером и типом данных его элементов. Размер массива должен быть достаточным для того, чтобы исключить переполнение стека. Если есть обоснованные оценки возможной глубины стека, то ответить на этот вопрос несложно; но так, к сожалению, бывает далеко не всегда, и тогда приходится отводить памяти значительно больше, чем фактически может быть использовано. Тип данных элементов, как правило, определить проще: если это стек возвратов, то элементы массива должны быть целого типа, если стек вычислений – то либо целого, либо вещественного типов; часто возникает необходимость в символьном типе (впрочем, символьный тип данных нередко реализуется опосредованно – через целочисленные коды символов).

Ниже приводится алгоритм реализации стека на основе массива (предполагается, что массив состоит из N элементов целого типа; индексация элементов массива начинается с 0):

```
[ИНИЦИАЛИЗАЦИЯ]
const MAXSIZE = N;
int stack [MAXIZE];
int top = -1;
```

```
[PUSH: ПРОТОЛКНУТЬ НОВЫЙ ЭЛЕМЕНТ В СТЕК]
if (top != MAXSIZE - 1)
then
увеличить значение указателя вершины на 1: top = top + 1
протолкнуть элемент в стек: stack [top] = element
else ошибка: print («Стек переполнен»)
```

```
[POP: ВЫТОЛКНУТЬ ЭЛЕМЕНТ ИЗ СТЕКА]
if (top >= 0)
then
вытолкнуть элемент из стека
уменьшить значение указателя вершины стека на 1: top = top -1
else ошибка: print «Стек пуст»
```

Новые элементы сохраняются в массиве в позициях с индексами 0, 1, ..., MAXSIZE – 1. Изначально значение указателя вершины стека top равно –1, что позволяет контролировать стек на пустоту – в непустом стеке значение top либо 0, либо положительно. При выталкивании элемента из стека изменяется только значение указателя вершины; сам элемент фактически остается в массиве, но доступ к нему посредством указателя вершины top отсутствует.

Из алгоритма следует, что работа со стеком на основе массива сводится к аккуратному манипулированию значением указателя вершины стека top. Этот метод реализации позволяет быстро получить доступ к любому элементу стека (хотя понятно, что этой возможностью не стоит злоупотреблять). Метод реализации стека на основе массива прост, эффективен и легко реализуем практически на любом языке программирования, но ему присущ ряд недостатков, которые часто делают его непригодным для решения некоторых важных задач:

- верхняя граница массива должна быть определена заранее. На практике это означает, что программист вынужден рассчитывать на наихудший вариант и резервировать памяти гораздо больше, чем фактически может потребоваться;
- массив может хранить данные только одного типа (например, только числа или только символы). Обычно этого вполне достаточно, но не всегда.

Именно такой способ реализации стека был использован в интерпретаторе расширенной стековой машины, о котором мы говорили во второй части книги. Программный код находится в файле *StackMachine.java* (см. приложение С).

Реализация стека на основе связанного списка

Теперь мы поговорим о втором способе реализации стека – на основе связанного списка. Этот метод позволяет обойти ограничения, присущие методу реализации на основе массива:

- нет необходимости заранее резервировать память для проталкивания в стек новых элементов – она выделяется динамически;
- после того как элемент выталкивается из стека, занимаемая им память может быть освобождена и возвращена в свободную память (разумеется, при надлежащей реализации);
- в связанных списках проще обеспечить реализацию элементов стека так, чтобы каждый элемент мог хранить данные различных типов.

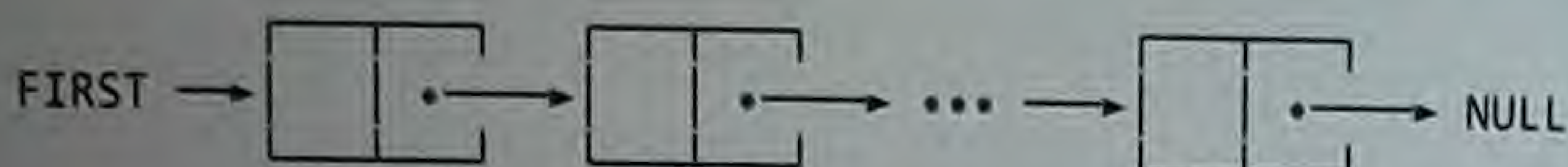
Конечно, у этого метода есть и свои недостатки:

- более высокая сложность реализации;
- необходимость выделения дополнительной памяти для указателей;
- меньшая скорость доступа к произвольным элементам списка, по сравнению с массивом (приходится просматривать каждый элемент, продвигаясь от элемента к элементу по указателям).

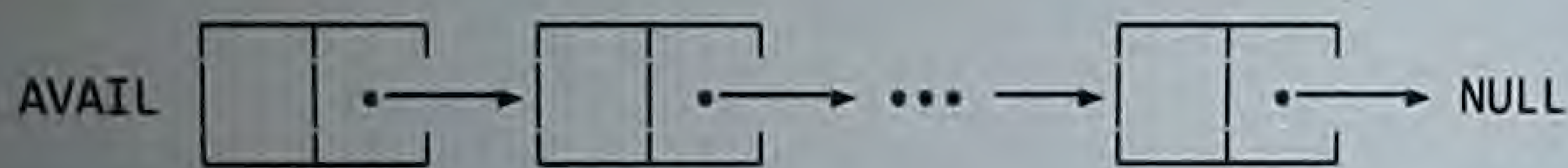
Алгоритм реализации стека на основе связанного списка следует в основных чертах описанию, предложенному Дональдом Кнутом; при этом используется следующая терминология: элемент связанного списка называется узлом («node»), состоящим из двух частей – поля данных («info») и поля связи («link»):



Поле данных предназначено для хранения данных, а поле связи указывает (адресует) предыдущий узел (т. е. узел, добавленный к списку перед текущим). Последний элемент списка (т. е. фактически вершина стека) адресуется указателем, обозначаемым как «FIRST»; поле связи последнего элемента указывает на пустой указатель «NULL» (работая с диаграммами, следует быть внимательным: вершина стека находится слева, а дно – справа):



Прежде всего необходимо обсудить, как выделяется память для нового узла. Память выделяется из списка свободных узлов, который обычно также организован в виде стека. Список свободных узлов («пул памяти») – это своего рода набор «заготовок»: если поле данных info стека хранит целые числа, то будет выделено одно количество памяти, если вещественные – другое, если строки – третье. Таким образом можно выделить столько памяти, сколько необходимо (в отличие от массива, элементы которого всегда одного типа), и в стеке будут одновременно находиться узлы с различными типами (и объемами занимаемой памяти) поля info. На рисунке список свободных узлов памяти выглядит практически так же, как и стек (указатель «AVAIL» содержит адрес последнего элемента списка свободных узлов):



Для поля связи link для всех узлов стека обычно выделяется одно и то же количество памяти, а именно столько, сколько необходимо для хранения адресов памяти.

Ниже приводится алгоритм добавления нового элемента в стек. В дополнение к указателю FIRST алгоритм использует вспомогательный указатель «NEW», предназначенный для хранения адреса узла, который станет новой вершиной стека.

```
[ПРОВЕРИТЬ НАЛИЧИЕ ДОСТУПНЫХ УЗЛОВ ИЗ СПИСКА СВОБОДНЫХ УЗЛОВ]
Если AVAIL равен NULL, то вывести сообщение «Список свободных уз-
лов пуст» и завершить работу алгоритма
[ПОЛУЧИТЬ АДРЕС ПЕРВОГО СВОБОДНОГО УЗЛА В СПИСКЕ СВОБОДНЫХ УЗЛОВ]
NEW = AVAIL
[ИСКЛЮЧИТЬ ТОЛЬКО ЧТО ВЫДЕЛЕННЫЙ УЗЕЛ ИЗ СПИСКА СВОБОДНЫХ УЗЛОВ]
AVAIL = LINK (AVAIL)
[СВЯЗАТЬ УКАЗАТЕЛЬ NEW СО СТЕКОМ]
INFO (NEW) = <данные>
LINK (NEW) = FIRST
FIRST = NEW
```

Алгоритм удаления узла из стека является по существу «перевернутой» версией алгоритма добавления нового элемента в стек:

```
[СТЕК ПУСТ?]
Если FIRST равен NULL, то вывести сообщение «Стек пуст» и завер-
шить работу алгоритма
[ИСКЛЮЧИТЬ ПЕРВЫЙ ЭЛЕМЕНТ СПИСКА (ВЕРШИНУ СТЕКА)]
FIRST = LINK (FIRST)
[ВЕРНУТЬ ИСКЛЮЧЕННЫЙ ИЗ СТЕКА УЗЕЛ В СПИСОК СВОБОДНОЙ ПАМЯТИ]
LINK (FIRST) = AVAIL
AVAIL = FIRST
```

Обратите внимание, что удаляемый из стека элемент возвращается туда же, откуда он был ранее выделен (т. е. в список свободных ячеек памяти). Если этого не сделать, а просто изменить поле связи в списке, реализующем стек, то исключаемый элемент окажется «мусором», а ссылка на него «повиснет». Это рано или поздно приведет к ситуации, когда памяти в целом более чем достаточно, но список свободных узлов исчерпан.

Мы не приводим программной реализации этого алгоритма – многочисленные примеры на различных языках программирования (прежде всего C и C++) легкодоступны как в сети Интернет, так и в многочисленных учебниках по структурам данных.

Заинтригованный читатель, не боящийся трудностей, может попробовать свои силы, реализовав этот алгоритм в виде программы для расширенной стековой машины (это действительно сложная задача, так что за нее лучше браться только после того, как будет приобретен достаточный опыт программирования более простых задач).

Приложение В

Язык Forth

В этом приложении мы приведем очень краткий обзор самого, пожалуй, известного стекового языка программирования – Forth. Это не описание, а именно обзор, поэтому для настоящего понимания, что представляет собой Forth, лучше обратиться к библиографии, где указаны некоторые полезные источники, включающие в себя многочисленные примеры.

Язык Forth был предложен в конце 60-х годов прошлого века Чарльзом Муром (США). В то время Ч. Мур занимался разработкой программного обеспечения управления телескопами. Ни один из существующих языков программирования не был достаточно эффективным для решения таких задач, и Муру пришлось искать другие решения. Результатом этих поисков стал Forth.

Очень быстро Forth приобрел огромную популярность и стал использоваться для разработки программ не только в астрономии, но и в других областях. Пик популярности Forth пришелся на начало и середину 80-х годов, после чего он был вытеснен другими языками программирования. Но как средство разработки Forth не исчез; Forth продолжает развиваться и используется при разработке программного обеспечения встроенного оборудования, управлении станками, роботами, медицинскими приборами. Forth оказал и продолжает оказывать сильное влияние при разработке системного программного обеспечения (трансляторы и операционные системы). Небольшой пример: Forth используется NASA для разработки программного обеспечения космических полетов (в частности, марсоходов), где чрезвычайно высокие требования к надежности и компактности.

Forth, в отличие от других языков высокого уровня, предоставляет программисту полный доступ ко всем составляющим компьютера – памяти и внешним устройствам; он не «прячет» от программиста ничего, что другие языки программирования тщательно оберегают. Это

позволяет создавать на Forth программы, почти не уступающие по эффективности программам, написанным на ассемблере. Но, в отличие от ассемблера, программы на Forth, как правило, более ясные и понятные.

Forth исключительно компактен и нередко может быть размещен в памяти микроконтроллеров. Ядро языка (в зависимости от реализации) составляет от нескольких килобайт до десятков килобайт, что резко контрастирует с другими языками программирования, для реализации которых порой требуются десятки и сотни мегабайт.

В отличие от других языков, Forth автономен и не нуждается в операционной системе; его можно запускать на «голом» оборудовании. Forth использовался для разработки BIOS и загрузчиков операционных систем. Идеи и концепции, впервые появившиеся в Forth, оказались чрезвычайно живучими.

Сейчас Forth занимает роль нишевого языка программирования, и о нем знают, к сожалению, немногие (а используют и того меньше). Но Forth не исчез и в обозримом будущем не исчезнет: он слишком хорош, чтобы остаться только в истории. Следы влияния Forth обнаруживаются повсюду; он настолько глубоко проник в современное программирование, что теперь от него так просто не «избавиться».

Основной структурной единицей программы на языке Forth является *слово*. Слово – это, в сущности, то, что в других языках называется подпрограммой. Для определения слова используется следующий простой синтаксис:

```
: имя_слова тело_слова ;
```

Здесь двоеточие (:) – т. н. *определяющее* слово, с которого начинается определение нового слова. Следом идут имя этого слова и список операций, составляющих тело слова. Завершает определение слова другое слово, а именно – точка с запятой (;). Разделителем между двоеточием, именем, телом и точкой с запятой является пробел. Операции, составляющие тело слова, в свою очередь, также являются словами, и они точно так же должны разделяться пробелами. Их (пробелов) может быть сколько угодно, но хотя бы один должен присутствовать.

Вот небольшой пример определения нового слова:

```
: inc 1 + ;
```

Здесь мы (после двоеточия) определили новое слово с именем inc. Это слово увеличивает значение своего аргумента на 1. Определение слова завершается точкой с запятой. Все элементы определения разделяются пробелами (в том числе перед словом ;, оканчивающим определение).

Но как этому слову передается аргумент, над которым нужно выполнить операцию увеличения на 1? Ответ может обескуражить – никак: явная передача аргументов в Forth отсутствует. Все данные (и аргументы в том числе) в Forth передаются через *арифметический стек* (или *стек данных*). Каждое слово в программе на Forth обращается к этому стеку за аргументами и оставляет в этом же стеке результаты.

Таким образом, если на вершине стека перед вызовом слова inc находилось число 12, то после того, как слово закончит свою работу, на вершине стека будет уже число 13.

Поскольку слова работают с данными, находящимися в арифметическом стеке, в языке существует предопределенный набор встроенных слов, некоторые из которых нам уже знакомы:

```
DUP, DROP, OVER, SWAP, ...
```

а также целый ряд других.

Разумеется, имеется целый ряд арифметических операций: сложение, вычитание, умножение и т. д. Их мнемоника в большинстве случаев совпадает с привычной математической (+, –, * и т. д.). По существу, рассмотренные нами каноническая и расширенная стековые машины являются прямыми наследниками Forth (хотя и очень сильно упрощенными).

Некоторые слова определены в ядре Forth и являются встроенными в язык; остальные – т. н. пользовательские – определяются программистом.

Определения слов, как встроенных, так и определенных программистом, собираются и хранятся в специальных структурах – *словарях*. Существуют системные словари, включающие в себя встроенные слова ядра языка Forth. Остальные словари – пользовательские. Вызов слова приводит к тому, что Forth-система ищет определение слова в словаре и исполняет его.

Что действительно делает Forth уникальным – так это способ хранения слов (и встроенных, и определенных пользователем) в словарях. Для этого используется одна из разновидностей т. н. *шитого кода*, при котором определения слов как бы нанизываются на нитку друг за другом, подобно бусинам. Слова, определенные позже, могут использовать слова, определенные раньше; таким образом, разработка программ на Forth ведется снизу вверх, от слов более низкого уровня к словам более высокого уровня. При определении новых слов мы пользуемся ранее определенными словами и как бы «склеиваем» их в одну конструкцию – новое слово. Эта операция носит название «конкатенация», и поэтому Forth часто называют конкатенативным языком (в эту ка-

тегорию также входят такие известные языки, как PostScript и Factor, которые, по существу, являются расширениями Forth).

Forth располагает развитыми средствами управления. Условная конструкция имеет следующий синтаксис:

```
IF <часть-тогда> ELSE <часть-иначе> THEN
```

или

```
IF <часть-тогда> THEN
```

(IF, ELSE и THEN – также слова). Слово IF берет из стека логическое значение и, если это истина, выполняет <часть-тогда>; в противном случае <часть-иначе> (эта часть может отсутствовать). Логическое значение истины представлено любым ненулевым значением; логическое значение лжи – нулевым значением. Таким образом, если на вершине стека не 0, то будет выполнена <часть-тогда>, иначе <часть-иначе>.

Существует несколько встроенных в ядро Forth слов, вырабатывающих логическое значение: >, <, =, 0=, 0> и 0<. Их назначение вполне очевидно.

Кроме того, в Forth имеются и циклические конструкции. Первая конструкция – цикл с условием:

```
BEGIN <часть-begin> WHILE <часть-while> REPEAT
```

или

```
BEGIN <часть-begin> UNTIL
```

Вторая конструкция – цикл со счетчиком

```
DO <тело> LOOP
```

Циклические конструкции Forth до определенной степени напоминают циклы while и for, хорошо знакомые по другим языкам программирования. Условия продолжения и счетчики циклов хранятся в стеке данных и в стеке возвратов.

В Forth имеется возможность определения констант и переменных, например следующим образом:

```
0 CONSTANT FALSE
1 CONSTANT TRUE
```

(здесь, очевидно, определяются две константы, представляющие логические значения лжи и истины соответственно). Теперь, вместо малозначащих чисел 0 и 1 в условных и циклических конструкциях можно использовать более понятные FALSE и TRUE.

```
VARIABLE age
```

А в этой конструкции определяется переменная с именем age (возраст). Когда в каком-нибудь слове будет использована переменная age, в стек данных будет помещен адрес этой переменной (именно адрес, но не значение, которое хранится по этому адресу).

Получить значение переменной позволяет слово @ (вспомните описание канонической стековой машины из первой части книги):

```
age @
```

после чего в стеке окажется значение переменной age. Присвоить новое значение переменной позволяет слово ! (опять же, обратитесь к описанию канонической стековой машины), которое сохраняет значение на вершине стека в переменной с именем age:

```
age !
```

Небольшой пример покажет, как работают слова @ и !. Пусть переменной age соответствует адрес 2500, и пусть по этому адресу хранится число 42. Тогда следующая последовательность слов увеличит значение по адресу age до 43:

```
Слово          Стек после выполнения слова
age                    ├ 2500
@                      ├ 42
1                      ├ 42 1
+                      ├ 43
age                    ├ 43 2500
!                      ├
```

Все определения констант и переменных также транслируются Forth-системой и сохраняются в словаре.

Разумеется, в Forth есть слова для работы с памятью, со строками, слова для организации ввода/вывода, а также слова для работы со словарем и создания собственных определяющих слов (помимо двоеточия). Все это обеспечивает Forth необыкновенную гибкость, лаконичность и компактность, правда, за счет некоторой потери наглядности. Без некоторой привычки читать программы на Forth нелегко, но это – несущественное препятствие, которое устраняется после нескольких уроков и упражнений.

На этом мы завершаем обзор языка программирования Forth и предлагаем, не откладывая в долгий ящик, попробовать язык в деле.

Приложение С

Стек и современные языки программирования

Реализации многих современных языков программирования базируются на стеке. Вернее будет сказать, что объектный код, получающийся в результате трансляции исходных кодов программ на этих языках программирования, представляет собой не что иное, как инструкции для определенной стековой машины.

Самыми, пожалуй, известными примерами таких языков являются языки Java и C#.

Исходные коды на языке программирования Java транслируются в байт-код JVM (Java Virtual Machine). Аналогично исходные коды на языке программирования C# (и других языков, поддерживаемых платформой .NET) транслируются в байт-код CIL (Common Intermediate Language). И JVM, и CIL представляют собой стековые машины.

После трансляции исходных кодов программ в байт-коды последние могут быть исполнены (интерпретированы) соответствующими виртуальными машинами.

В байт-коды JVM и CIL могут быть оттранслированы не только программы на Java и C# соответственно. Поскольку спецификации этих виртуальных машин опубликованы и имеются все необходимые инструменты, то для любого языка программирования при желании может быть написан транслятор в нужный (целевой) байт-код. Таким образом, программист может использовать любой язык программирования, для которого имеется транслятор в байт-код. Список таких

реализаций весьма велик (несколько десятков), и он постоянно продолжает пополняться. Скажем, существуют трансляторы в байт-код JVM таких популярных языков программирования, как C, Lisp, Forth (да-да, и Forth тоже), JavaScript, Lua, Pascal, PHP, Python и Ruby. Для платформы .NET также существует внушительный список языков программирования.

Что дает такая трансляция в байт-код? То, что для его исполнения достаточно одной виртуальной машины, а в проектах можно использовать самые разные языки программирования, не особенно беспокоясь о том, как связывать полученные после трансляции объектные коды друг с другом.

Это – превосходная технология, но, как всегда, ничего не дается даром. Байт-коды при исполнении обычно оказываются более медленными, чем объектные (машинные) коды для конкретного процессора. Для решения проблем с производительностью широко используются технологии JIT (Just In Time, или компиляция на лету), при которой байт-код (весь или частично) преобразуется в машинный код непосредственно во время работы программы. JIT-технологии значительно повышают скорость исполнения, приближая ее к скорости машинного кода.

Если вы решите попробовать свои силы в создании трансляторов, генерирующих байт-коды для выбранной виртуальной машины, вы должны быть готовы к трудной работе: спецификации виртуальных машин далеко не просты, и на их понимание придется потратить немало сил и времени. Но дорогу осилит идущий.

Приложение D

Исходные коды транслятора и интерпретатора

Ниже приведены исходные коды транслятора и интерпретатора расширенной стековой машины, которая была описана во второй части книги. Для трансляции исходных кодов необходим JDK версий 7/8 (после несущественных исправлений исходных кодов можно использовать и Java 6, но мы не рекомендуем этого делать).

Некоторые указания для тех, кто мало знаком (или вовсе не знаком) с языком программирования Java и у кого мало опыта компиляции исходных кодов java-программ из командной строки.

Прежде всего сохраните исходные коды в файлах с расширением java; всего должно быть пять файлов:

- *ToyStackMachine.java*;
- *OpcodeTable.java*;
- *SymbolTable.java*;
- *Assembler.java;*
- *StackMachine.java*.

Все пять исходных файлов должны находиться в пакете (т. е. в каталоге) *toystackmachine*.

Регистр символов в именах имеет значение (на этом месте, к сожалению, «спотыкаются» почти все начинающие программировать на Java). Подробное описание всех файлов приведено во второй части книги.

Теперь исходные коды нужно откомпилировать. Если вы не используете IDE, то выйдите из каталога toystackmachine и введите команду:

javac toystackmachine/.java*

(обратите внимание на команду – она должна вводиться точно так, как мы указали). Если вы используете IDE, то поищите, как можно откомпилировать ваш проект (как правило, такая кнопка располагается на самом видном месте).

Если при компиляции не будет ошибок, то в каталоге toystackmachine будут сформированы class-файлы, содержащие байт-коды.

Теперь приложение можно запустить следующей командой:

java toystackmachine.ToyStackMachine

(опять-таки, будьте внимательны – команда должна быть введена именно так, как мы указали). В консоль будет выведено уже знакомое приглашение:

```
++++++++++++++++++++++++++++++++
+ Assembler for ToyStackMachine +
++++++++++++++++++++++++++++++++
Source file name:
```

Вот и все: введите имя файла исходного кода программы расширенной стековой машины (по умолчанию эти файлы должны располагаться вне каталога toystackmachine) и «насладитесь» результатами долгой и трудной работы.

ДАЛЕЕ
СЛЕДУЮТ ИСХОДНЫЕ КОДЫ
ВСЕХ ПЯТИ ФАЙЛОВ, СОСТАВЛЯЮЩИХ
ПРИЛОЖЕНИЕ

ToyStackMachine

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Вторников Алексей Анатольевич

Стек, или Путешествие туда и обратно

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 13.125. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

Вторников Алексей Анатольевич

Стек, или Путешествие туда и обратно

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 13.125. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф



Стек,
или Путешествие туда и обратно

Книга посвящена простой и удивительно элегантной структуре данных – стеку.
Описаны скобочные структуры, подпрограммы (в том числе рекурсивные), передача параметров, разбор и вычисление выражений, распознавание последовательностей символов. Также рассмотрено описание устройства и реализация простой, но достаточно мощной стековой машины; приведены многочисленные примеры программ, а также список задач, в том числе нетривиальных.

Издание предназначено прежде всего пытливым старшеклассникам, студентам вузов, а также тем, для кого программирование – хобби.

На сайте издательства www.dmkpress.com содержатся дополнительные материалы, среди которых исходные коды [illegible] транслятора стековой машины (на языке Java).

Удачи – и пусть стек станет вашим настоящим другом и помощником!

